International Journal of Innovative Computing, Information and Control
Volume **8**, Number **7(A)**, July **2012**

ICIC International ©2012 ISSN 1349-4198
pp. 4771–4796

EVOLUTIONARY ALGORITHMS TO SOLVE LOOSELY CONSTRAINED PERMUT-CSPS: A PRACTITIONERS APPROACH

Luis de-Marcos, Antonio García-Cabot and Eva García

Computer Science Department
University of Alcalá
28871, Alcalá de Henares, Madrid, Spain
{ luis.demarcos; a.garciac; eva.garcial }@uah.es

Received March 2011; revised July 2011

Abstract. *Permutation constraint satisfaction problems (permut-CSPs) can be found in many practical applications, wherein most instances usually have a low density of constraints. This paper explores two evolutionary approaches to solve this kind of problem from a practical perspective. A test case that captures the main characteristics present in real world applications is used to design and test the performance of a PSO agent and a GA agent, which are also systematically tuned to determine the best configurations using statistical analysis. We conclude that the PSO agent provides a better fit to the nature of loosely constrained permut-CSPs, resulting in better performance. This paper focuses on the trade-off between development costs (including tuning) and the performance of both evolutionary algorithms, and aims to help practitioners choose the best algorithm and configuration for these problems.*

Keywords: Constraint satisfaction, CSP, PSO, Genetic algorithm

1. **Introduction.** A constraint satisfaction problem (CSP) is a problem composed of a set of variables that must be given a value and a set of constraints that restrict the values that those variables can take. Thus the aim of a CSP-problem solver is to find an assignment for all the variables satisfying every constraint [1]. If all the solutions of a CSP are permutations of a tuple, then the CSP is said to be a permut-CSP. A permut-CSP is Loosely Constrained (LC-permut-CSP) when the set of constraints does not reduce to an extreme the number of feasible solutions. That usually occurs because the density of constraints is low and as a result problems are easier to solve than harder instances which have very few, if any, feasible solutions. So far, these problems have not received much interest from the research community because they are easy to solve and many different methods may be employed. Studies usually focus on problems which are more difficult to solve and pose greater challenges. As an example, much attention has been paid to NP-Hard permut-CSPs such as the N-queens problem, the graph coloring problem, or the knapsack problem. There is no doubt that it is worth studying better ways to solve these problems because these solutions can be employed to solve real world problems in many domains. Nevertheless, LC-permut-CSPs also have many applications in real world problems including planning, logistics, sequencing, resource allocation, networking, scheduling and cryptography [2-4], just to mention a few, in a wide variety of domains such as medicine [5], software engineering [6,7], education [8,9], manufacturing [10] and telecommunications [11]. LC-permut-CSPs are thus important for many practitioners and engineers. Hard CSPs are also important but almost all the resources have been devoted to them, apparently leaving LC-CSPs aside, besides which traditional CSPs research focuses on algorithm performance over any other issue producing algorithms that may

not be efficient in other respects such as the time they consume, or their difficulties in implementation or operation.

The usual way to approach and solve practical LC-permut-CSPs comprises the following steps: (1) review the current literature to select one or more appropriate methods, (2) implement and tune them, and (3) finally test them. This process is time consuming and these steps can become really cumbersome and difficult. For example, algorithm tuning can be very problematic since some techniques require many different parameters, and each parameter requires making several decisions concerning its possible values. Moreover, there are not only dependencies among parameters in terms of efficiency, but also different techniques to tune them, each one with its own advantages, disadvantages, complexity, performance, and so on. In today's world, many engineers are required to build fast and efficient solutions, but organizations usually do not have the resources required to let those engineers thoroughly complete these tasks. The objective of this paper is to fill that gap and help practitioners select an efficient, but also reasonably easy to implement, solution to their problem along with a systematic method for tuning it in order to establish the best parameters when facing an LC-CSP. It is important to use methods that can be systematically employed to solve LS-CSPs efficiently. The aim is not just to solve the problem, nor to apply the best finely-tuned algorithm that offers the best performance, but to find a balance between complexity and efficiency.

From that perspective, our objective is to focus on algorithms that offer reasonable performance and that can be developed and tested using also reasonable resources. Two fairly recent evolutionary optimization methods have shown a good balance between performance and complexity in a wide variety of problems in different domains. Genetic algorithms were introduced in the mid 70s [12] and they simulate natural selection processes in order to find solutions to problems. Particle swarm optimization (PSO) was introduced in the mid 90s [13,14] as a new problem solver based on the foraging behavior observed in social insects like bees. Both offer all the aforementioned desired features along with an innovative approach to handle permut-CSPs. This paper aims to systematically test and compare these evolutionary techniques to solve LC-CSPs, simultaneously focusing on the trade-off between their efficiency, and their development (and tuning) requirements. To accomplish that, the paper is structured as follows. Section 2 reviews related work and defines the paper's contribution. Section 3 presents a mathematical characterization of the problem. Section 4 describes the designed test cases and the algorithms that were developed to solve them. Section 5 presents the results of the different experiments. Finally, Section 6 presents the conclusions and future research opportunities.

2. Related Works and Contribution.

Seminal works on the study of CSPs focus on binary CSPs [15,16]. They focus on exact methods (BFS, DFS) and heuristic methods (A*) to determine the solvability of random binary CSP instances and to analyze the features of the spaces of solutions. As a consequence, such approaches find all solutions to any given problem. Thus they are ineffective for our approach since the vast majority of LC-permut-CSPs have many solutions, all them being equally valid in many practical applications. Different metaheuristic population-based methods have also been adapted to deal with CSPs. Genetic algorithms that consider almost any variant of permutation and constraint have been presented and analyzed. Comprehensive reviews are presented in [17,18]. A version of the PSO algorithms that deals specifically with discrete binary variables was introduced by Kennedy [19]. A version for permut-CSPs is presented by Hu [20], comparing its performance in the N-queens problem against previous methods. All these approaches are adaptations of methods employed to solve problems that deal with continuous variables and spaces. Although they have also demonstrated good performances in solving permut-CSPs, they have been studied thoroughly to solve hard instances of permut-CSPs and little attention has been paid to their efficiency in solving loosely constrained instances, despite the fact that many practical applications and real world problems belong to this class of LC-permut-CSPs. Ant colonies, on the other hand, are an optimization method that has been specifically designed for discrete spaces. The suitability of this approach in solving CSPs has been addressed in the works of Schoofs and Naudts [21] and Solnon [22]. Solnon studied the performance of the Ant Colony Optimization (ACO) solver for a set of binary random generated permut-CSPs. Although she establishes a good testbench for comparison she does not compare her solver with any previous approaches. Solnon's work also combines the ACO solver with a local search in a hybrid development. Besides, all the methods mentioned here pay little attention to parameter definition and tuning which are highly important for efficiency.

Studies on parameter tuning and control are also related to our work. Eiben et al. [17] presented a comprehensive review and taxonomy. A review of this study reveals that there are many methods, each having its own set of advantages and disadvantages. More recent approaches propose tuning algorithms [23]. It can also be observed that many of these approaches focus on finding the best possible set of parameters (optimal solution) [24], while there is also evidence that nearly-optimal solutions can be found with much less effort [25]. Nearly-optimal sets of parameters are good for many practical applications. The benefit of an optimal solution in terms of performance is usually minimal and does not pay back for the effort that may be required to obtain it. Besides, studies on parameter control do not pay much attention to permut-CSPs and, to the knowledge of the authors, there are no studies on the effects of parameter control on practical applications of LC-permut-CSPs. This is important since the nature of spaces of solutions in permutation discrete landscapes is completely different.

This paper presents a framework for the systematical comparison of evolutionary algorithms. Our focus is on permut-LC-CSPs because they have received little attention up to date, but also because they are common in many real world applications. To develop such a framework requires: (1) characterization of the problem and defining the experimental test cases that capture the essence of permut-LC-CSPs, and (2) the defining, developing and testing of a method to compare different methods that enable a degree of flexibility sufficient to facilitate the addition of new algorithms to the testbench. We use genetic algorithms and the PSO as the initial solvers for testing our framework. As our focus is on practical applications and on the trade-off between the development cost and efficiency of the algorithms, hybrid methods are initially excluded as they will require programming two or more algorithms. We pay special attention to parameter tuning in such a way that we propose a statistical method that finds nearly-optimal sets of parameters, for the algorithms that are analyzed here, and that can be systematically employed to determine new configurations for different algorithms.

3. **Problem Characterization.**

The problem to be studied will be characterized as a constraint satisfaction problem (CSP). A CSP comprises a set of variables which must be given a value within a predefined domain and a set of constraints that restrict the values such variables can be given. Constraints are common in real life, and that is why these kinds of problem are useful as an abstract tool with which to model problems that include constraints.

Following Tsang [1], we will define a CSP as a triple (X, D, C) where $X = \{x_0, x_1, \ldots, x_{n-1}\}$ is a finite set of variables, D is a function that maps each variable to its corresponding domain $D(X)$, and $C_{i,j} \subset D_i \times D_j$ is a set of constraints for each pair of values (i, j) with $0 \leq i < j < n$. In order to solve the CSP all variables x_i in X must be assigned a value from its domain D, such that all constraints are satisfied. A constraint is satisfied when $(x_i, x_j) \in C_{i,j}$, and when (x_i, x_j) it said to be a valid assignment. If $(x_i, x_j) \notin C_{i,j}$, then the assignment (x_i, x_j) violates the constraint.

If all solutions from a CSP are permutations of a given tuple then the problem is a permutation CSP or PermutCSP. A PermutCSP is defined by a quadruple (X, D, C, P) where (X, D, C) is a CSP and $P = \langle v_0, v_1, \ldots, v_{n-1}\rangle$ is a tuple of $|X| = n$ values. A solution S of a PermutCSP must be a solution of (X, D, C) and a complete permutation of P.

LC-CSPs may be characterized as CSPs or PermutCSPs in this way. For example, consider the problem of ordering five tasks named 1, 2, 3, 4 and 5; the PermutCSP of which the only solution is the set $S = \{1, 2, 3, 4, 5\}$ (all tasks must be ordered) can be defined as:

$$\begin{aligned}
&X = \{x_1, x_2, x_3, x_4, x_5\}\\
&D(X_i) = \{1, 2, 3, 4, 5\} \quad \forall x_i \in X\\
&C = \{x_{i+1} - x_i > 0 : x_i \in X,\ i \in \{1, 2, 3, 4\}\}\\
&P = \langle 1, 2, 3, 4, 5\rangle
\end{aligned}$$

Many heuristic and meta-heuristic methods, evolutionary algorithms among them, require a fitness function to evaluate the potential goodness of each solution in order to compare them with other potential solutions. It is critical to choose a function that accurately represents the goodness of a solution and a bad choice may seriously affect performance [11]. When the problem domain does not provide an objective function, a common choice for CSPs is a standard penalty function [21].

$$f(X) = \sum\nolimits_{0 \le i < j < n} V_{i,j}(x_i, x_j) \tag{1}$$

where $V_{i,j} : D_i \times D_j \to \{0, 1\}$ is the violation function

$$V_{i,j}(x_i, x_j) = \begin{cases} 0 & \text{if } (x_i, x_j) \in C_{i,j} \\ 1 & \text{otherwise} \end{cases} \tag{2}$$

This fitness function works well if the constraint set C for the CSP has been accurately defined. In the previous example, the restriction set was defined as $C = \{x_{i+1} - x_i > 0 : x_i \in X,\ i \in \{1, 2, 3, 4\}\}$. A more accurate definition would be $C = \{x_i - x_j > 0 : x_i \in X, x_j \in \{x_1, \ldots, x_i\}\}$. If we consider the sequence $\{2, 3, 4, 5, 1\}$ the standard penalty function will return 1 if the first definition of C is used, while the returned value will be 4 if it uses the second definition. The second definition is more accurate because it returns a better representation of the number of swaps required to turn the sequence into the valid solution. Moreover, the first definition of C has additional disadvantages because some completely different sequences (in terms of its distance to the solution) return the same fitness value. For example sequences $\{2, 3, 4, 5, 1\}$, $\{1, 3, 4, 5, 2\}$, $\{1, 2, 4, 5, 3\}$ and $\{1, 2, 3, 5, 4\}$ will return a fitness value of 1. Opportunely, this question can be solved programmatically. A function that recursively processes all restrictions and calculates the most precise set of restrictions violated by a given sequence can be programmed and called when the input sequence is initialized. The user will usually define the minimum number of constraints necessary and the system will compute the most accurate set in order to facilitate convergence, so user obligations can simultaneously be reduced.

We have characterized the problem as a CSP because in this way we have a powerful mathematical apparatus to manipulate the problem. It also provides the mechanism to abstract the problem from the details of implementation of each specific case. Finally it also provides a fitness function that is well defined and easily computable. This is a requirement for any heuristic method. A further two constraints have been imposed to develop our work. Firstly, the problem is defined as a permutation problem, and secondly it is supposed to be loosely constrained. These decisions are conditioned by the kind of practical applications that should be susceptible to using the results developed here. The authors believe that many real applications share the features of these problems. In the next section, we propose a test case that captures the essence of such problems providing appropriate justification.

4. **Experimental Tests.**

The initial stages of our research comprise the selection of a representative test case along with the algorithms and configurations that will later be tested. Both elements, and the rationale underlying every decision, are presented in this section.

4.1. **Test case.**

One important concern was to select a test case that appropriately resembles the features that can be found in as many real applications of LC-permut-CSPs as possible. In order to achieve that, we performed a literature review searching for common patterns in CSPs which are applied in different domains, including scheduling, logistics, planning and sequencing. Despite the fact that this is a difficult task because every approach included its own details and nuances, we present here a case which we think represents their main features. Our focus is restricted to applications of combinatorial optimization methods. The final test case included 24 elements grouped as follows:

- A set of elements which must precede every element from the other sets (7).
- A set of elements that must be given in a predetermined order (5).
- A set of compulsory elements that must be performed with no predefined constraints concerning its internal order (7).
- A set of free introduced elements. There are no special constraints with other groups, with the exception that they must be completed after the elements of the first group (5).

We will now present an allegory to turn this test case into a hypothetic task scheduling problem. The first group of elements will represent a set of pre-tasks that must be completed before any other task. If our task scheduling problem refers to a production factory we may think in tasks related with the starting and tuning of different machines. The second group represents a set of tasks that must be completed in a predefined and fixed order. We may think in different kinds of industrial processes aimed at producing different goods. The third group comprises a set of tasks that must be performed during the operation but in no particular order at all; they can be 'interspersed' throughout the sequence. The fourth and final group comprises a set of tasks randomly selected for different purposes. We added them because they are common in many applications. For example during project task scheduling (see [26,27], for example) this kind of task may be introduced to assign resources (usually people) that have a cost and otherwise remain unused. Another example relates to curriculum sequencing (applied in Intelligent Tutoring Systems [28,29]), where this kind of task represents optional or free choice subjects or content. Following this allegory, in the remainder of the text we will call the four groups pre-tasks, ordered tasks, post-tasks and free tasks for the sake of simplicity.

So far we have four different groups of elements, but additional decisions need to be made concerning the test case. First of all it is necessary to define how many elements are placed in each group. Analyzing the aforementioned studies we found problems that deal with 10 up to 100 elements (and even more than 200 in some experimental test cases), but in most cases the number does not exceed 25 elements. We take this value as an average number for our test case. With respect to the number of elements in each set, we decided to include 7 pre-tasks, 5 ordered tasks, 7 post-tasks and 5 free tasks in order to have balanced groups. Thus, the final number of elements in the test case is 24.

A further analysis of the empirical data revealed that problems also have two other sets of constraints. These are found between elements inside every group, and also between elements from different groups that were not specified in the previous model. These constraints seem somehow to be arbitrary, and it was impossible for us to discern any pattern within them. However, noticing the fact that they exist regularly is a fact of their importance, so we consider it necessary to establish a method to include them. We computed them for a number of sample problems and weighted them according to different (sample problem) sizes to get a measure of their density. Our final decision is to randomly create up to five internal arbitrary constraints on pre-tasks, post-tasks and free tasks groups ($p = .85$). Another five arbitrary constraints are created to further constrain tasks belonging to different groups ($p = .85$). Random numbers are generated to decide the final number of constraints and the tasks that will be created. Thus every test case had 18 to 38 constraints. Eighteen constraints exist due to the nature of the problem: The 7 pre-tasks must precede ordered tasks, the 5 ordered tasks must be ordered (4 constraints), and the 7 post-tasks must succeed any pre-task. Twenty additional constraints may be randomly included. Their inclusion probability is 0.85 and that will mean an average test case will have 35 constraints. Duplication of constraints is not allowed. If each new random constraint is identical to a previous existing one then it is generated again. Cycle checking is also required to exclude problems with no solution. Incomplete approaches cannot detect inconsistency, so we consider only feasible instances. Due to the random component that exists in the test cases generation, the same set of problems is used for every algorithm and configuration that we subsequently tested. 100 different test cases were generated using that method and stored in order to be used for experimental testing.

Please note that each constraint involves exactly two variables. They are binary constraints and thus we will be dealing with randomly generated binary permut-CSPs. A class of randomly generated binary CSPs is characterized by the 4-tuple $\{n, m, p_1, p_2\}$ [22]. m is the number of variables and n is the number of values in each variable domain. p_1 is the constraint density. It is the portion of the $n \cdot (n-1)/2$ constraints in the graph. Considering that the average test case has 35 constraints, p_1 will be .127. p_2 is a measure of the tightness of constraints. It determines the number of incompatible pairs of values for each constraint.

The problem may seem under-constrained, which would result in it being easy to solve for any solver. To further justify our decisions we will turn now to studies on phase transition. These works focus on the transition in solvability that can be observed when one parameter changes. Transitions have an easy-hard-easy structure and the region in which changes occur is called the mushy region [30]. It determines the area where most of the hard instances exist, and therefore it may be considered as the most promising area to test problem-solvers. The mushy region is wider when smaller values of p_1 are employed and it is narrower when the constraint density is higher [16]. That means that with a lower value of p_1 it is more likely to create hard instances when a random problem generator is employed. Prosser further reports that in his experiments when $p_1 = .1$ there is a higher variability in search effort, even well before the phase transition. We find this to be a very desirable characteristic to test the performance of problem solvers since harder instances will be present among a fair amount of easy instances. As the density of the constraint graph increases, the search effort for the hardest instances also increases but variability diminishes, the mushy region narrows and consequently hard instances are more difficult to find. A similar behavior is observed as n increases. The mushy region narrows (and the search effort increases exponentially). We then need to find a balance between computational cost and solvability. As we previously explained, we set $n = 24$ to be able to run a sufficient number of tests to have statistical significance. As for the domain size (m) it is set as $m = n$ because we have a permut-CSP in which all solutions are complete permutations of a given tuple. Prosser study also asserts that the width of the mushy region does not change significantly as the domain size varies, so this setting does not have any significant influence on the final results.

The selected test case may also seem to be quite generic and simple because its main features were abstracted from a series of practical studies. In any case, it is important to note that it is easy to analyze while it is also quite flexible, adaptable and extendable to comply with other different configurations. We also think that it provides a good ground to gain a better insight into how practical LC-permut-CSP are stated and how different problem solvers will perform when dealing with them. Works that study the nature of problems are numerous but they are mainly theoretical studies focused on very limited and specific instances. [15] is an example in which we find a characterization of the graph coloring problem with many practical limitations. At the same time such works point towards the necessity "to build random problem generators that can be tuned to fit the unique characteristics of real world problems such as job-shop and telescope scheduling, classroom timetabling and some numerical computations" [30]. We have tried to design a test case that is closer to real world applications looking at the theoretical foundations whenever possible.

4.2. **Algorithms.** General search methods for CSP solving include backtracking, lookahead, backchecking and backmarking, stochastic search methods and hybrid algorithms [1]. A more recent trend looks for inspiration in nature and natural processes to build problem solvers targeting a wide range of domains. Among these approaches, genetic algorithms and particle swarm optimization have been successfully applied to solve many different problems in different domains. [31] and [4] present detailed lists of applications for each approach. We have designed one version of each algorithm to deal with permut-CSPs. Both are presented in the following subsections.

4.2.1. *PSO algorithm to solve permut-CSPs.* Particle swarm optimization (PSO) is an evolutionary computing optimization algorithm. PSO imitates the behaviour of social insects like bees. A randomly initialized particle population (states) flies through the solution space sharing the information they gather. Particles use this information to dynamically adjust their velocity and cooperate towards finding a solution. The best solution found: (1) by a particle is called *pbest*, (2) within a set of neighboring particles is called *nbest*, (3) and within the whole swarm is called *gbest*. *gbest* is used instead of *nbest* when a particle takes the whole population as its topological neighbors. The goodness of each solution is calculated using a fitness function.

Original PSO [13,14] is intended to work on continuous spaces. A discrete binary version of the PSO was presented in [19]. This version uses the concept of velocity as the probability of changing a bit state from zero to one or vice versa. A version that deals with permutation problems was introduced in [20]. In this latter version, velocity is computed for each element in the sequence, and this velocity is also used as the probability of changing the element, but in this case the element is swapped, establishing its value to the value in the same position in *nbest*. Velocity is updated using the same formula for each variable in the permutation set, but it is also normalized to the range 0 to 1 by dividing each element in the set by the maximum range of the particle (i.e., maximum value of all). The mutation concept is also introduced in this permutation PSO version. After updating each particle's velocity, if the current particle is equal to *nbest* then two randomly selected positions from the particle sequence are swapped. In [20] is also demonstrated that permutation PSO outperforms genetic algorithms for the N-Queens problem. So we decided to try PSO as the first optimizer to solve LC-CSPs.

Each particle shares its information with a, usually fixed, number of neighboring particles to determine *nbest* value. Determining the number of neighboring particles (the neighbor size) and how neighborhood is implemented has been the subject of research in an area called sociometry. Topologies define structures that determine neighborhood relations, and several of them (ring, four cluster, pyramid, square and all topologies) have been studied. Evidence demonstrates that fully informed approaches outperform all other methods [32]. The fully informed approach prompts using 'all' topology and a neighborhood size equal to the total number of particles in the swarm. That means that every particle is connected with all other particles when *nbest* values are calculated, hence *gbest* is always equal to *nbest*.

One important PSO advantage is that it uses a relatively small number of parameters compared with other techniques such as genetic algorithms. Nonetheless, a great deal of literature on PSO parameter exists to help practitioners establish them [33,34]. Among this, [20] established the set of parameters in such a way that PSO works properly for solving permutation problems. As a result, we decided to follow their recommendations, and parameters are set as follows: Learning rates $(c1, c2)$ are set to 1.49445 and the inertial weight (w) is computed according to the following equation:

$$w = 0.5 + (rand()/2) \tag{3}$$

where $rand()$ represents a call to a function that returns a random number between 0 and 1. The inertial weight (w) governs what amount of the velocity is determined from the particle's current inertia and how much is computed from learning. Learning rates represent the amount of information that each particle uses from the best position found so far by the particle (*pbest*) and by the whole swarm (*gbest*). $c1$ and $c2$ weight the importance given to local learning and social learning. Population size is set to 20 particles. Considering that a fair amount of research in PSO agrees on the point concerning parameter setting, and that the trade-off between algorithms' efficiency and its development time is an important premise for our research, it was decided to use these settings concerning parameters in all our tests. As the fully-informed version is used, it is not necessary to make any consideration concerning the size of the neighborhood.

During the initial agent development we found that in some situations the algorithm got stuck in a local minimum, not being able to find a feasible solution. For that reason, two modifications are envisaged in order to try to improve algorithm performance for LC-CSPs. The first change is to randomly decide whether the permutation of a particle's position is performed from *gbest* or from *pbest* ($p = 0.5$). In the original version, all permutations are conducted regarding *gbest*. The second modification consists in changing *pbest* and *gbest* values when an equal or best fitness value is found by a particle. In other words all particle's comparisons concerning *pbest* and *gbest* against the actual state are set to less than or equal ($<=$) because the fitness function has to be minimized. The original algorithm determines that *pbest* and *gbest* only change if a better state is found (comparisons strictly $<$). Code fragment 1 presents the algorithm code. The algorithm proceeds in an iterative manner after randomly initializing the positions and velocities of the particles. Until a solution is found, and for each particle, *pbest* and *gbest* are updated, the new velocity is computed and normalized and the particle's position is updated. Finally, mutation is performed.

The underlying idea is to increase the particles' mobility and to avoid quick convergence to local minima. Three elements are involved in the computation of the new velocity. These are the current inertia of the particle, *pbest* position and *gbest* position. The actual movement of the particle (which is indeed a permutation) is determined based on the *gbest* current position. If the final permutation is instead sometimes performed towards *pbest*

Code 1. PSO procedure to solve LC-CSP

```
PSO (input_sequence, w, c1, c2)
BEGIN
  SET population[0] = input_sequence
  Randomly INITIALIZE rest of the swarm positions
  Randomly INITIALIZE particles velocity in the range 1..lengh(input_sequence)
  DO
    FOR-EACH particle X {
      SET newfitness=fitnessValue(X)
      IF (newfitness <= gbest)
        SET gbest= X
      IF (newfitness <= pbest)
        SET pbest = X
      Calculate new velocity as
        $V_{new}$ = w * $V_{old}$ + c1 * rand() * (pbest - X) + c2 * rand() * (gbest - X)
      Normalize Velocity as
        $V_{norm}$ = $V_{new}$ / max($V_{new}$)
      Update particle value
        FOR-EACH v[i] in $V_{norm}$ {
          IF(rand() < v[i])
            IF(rand() < 0.5)
              swap X[i] for X[indexOf(X,pbest[i])]
            ELSE
              swap X[i] for X[indexOf(X,gbest[i])]
        END FOR-EACH
      Check Mutation
        IF (X = gbest) swap two random positions from X
    END FOR-EACH
  UNTIL(termination criterion is met)
END
```

we can have particles exploring different regions of the solution space. Otherwise (if all the permutations are performed towards *gbest*), all the particles could end up exploring the same basin which could finally lead the swarm to a local minimum which is not a feasible solution. This was the purpose of the first modification. With respect to the second modification, our proposal prompts changing the values of *pbest* and *gbest* as many times as possible to guide the particles towards new directions (these values are used in the new velocity computation). It should also be noted that we are facing a discrete optimization scenario and use a standard penalty function to determine the fitness of each state. That means that we can have many states which return the same fitness value. If *pbest* and (especially) *gbest* do not change for a long time, the swarm can stagnate and continue exploring one region that does not contain a feasible solution. If *pbest* and *gbest* values are updated every time that an equal or better fitness value is found, we are constantly forcing the particles' movement towards new directions and thus enlarging the search region. All these modifications are tested later in the experimentation phase.

4.2.2. *Genetic algorithm to solve permut-CSPs.* Genetic algorithms are an evolutionary computation technique that simulates the evolution of genes to solve problems. A randomly initialized population of individuals is created where each individual contains a coded state or solution (gene) to the problem. Then an iterative process of recombination, mutation and selection is used to evolve the population and, simultaneously, the solution. Individuals can be coded in many different ways. From the GAs initial conception [12], binary representations have been widely used and sometimes abused. A codification that accurately fits the nature of the problem is recommended and binary codes are not always the best choice. Thus, additional ways for representing individuals were researched and documented [35]. In this way integer, real and tree representations, among others, can be used. The representation critically affects operators' (recombination and mutation) choices. Usually each representation comes with its own and well known set of operators.

For permutation problems, a sequence of integer values, in which the repetition of each value is not allowed, is the most usual way to represent individuals. Standard typologies distinguish between *order problems* (e.g., job scheduling problem) and *adjacency problems* (e.g., travel salesman problem) [36]. A specific set of recombination and mutation operators already exists for each of these two kinds of problem. Most common mutation operators are swap mutation, insert mutation, inversion mutation and scramble mutation [37]. The usual choices for recombination operators include partially mapped crossover [38] and edge crossover [39] for adjacency problems; and order crossover [3] and cycle crossover [40] for order based problems. GAs that use specific representation and operators for handling permutations are called permut-GAs and can be employed to solve constraint satisfaction problems.

Besides representation and iteration operators, a GA requires ways for handling the population, and consequently the solution's evolution. Specific operations to select individuals must be performed at two instances during the execution of the algorithm: parent selection and survivor selection. Common mechanisms for selection that can be applied in both processes are fitness proportional selection [41], ranking selection [42] and tournament selection [43]. These operations rely on each individual's fitness and random mechanisms to decide the set of individuals selected for recombination or survival. For survivor selection, a replacement policy (or model) is usually set to further restrict the set of individuals that are allowed to be selected and the way in which the population evolves and grows. In a generational model the entire population is replaced with the new successors, and in a steady-state model [44] just a subset of the population is replaced. Additional features such as elitism, aged-based replacement strategies, or duplicate elimination policies can also be implemented.

Permut GA with order recombination, swap mutation and generational replacement with elitism is implemented in order to test its performance for solving LC-CSPs. In order crossover two random crossover points are drawn. The sequence between them is copied from the first parent to the offspring and then, starting at the second crossover point, the remaining positions are copied from the second parent. We choose order crossover because it is easier to implement and avoids the necessity of finding cycles inside sequences. In swap mutation, two random positions are drawn and their values are interchanged. Swap mutation is preferred since it is easier to implement and also because it is most commonly used and fits better with order problems. As regards the replacement strategy, generational replacement consists of replacing the whole population by the new one. It is usually preferred over steady-state models. Generational models, however, have an important drawback: most fitted individuals are destroyed on each iteration. This is because all individuals are replaced by the next generation. To mitigate this problem, elitist strategies are implemented to keep track of the best individuals found so far and to include them in the next. Code fragment 2 shows the basic procedure. After the population is randomly initialized, an iterative process is run until a solution is found. The iterative process is divided into two stages. Firstly selection, recombination and mutation are performed. Then replacement is carried out. Selection entails a generational replacement keeping the best individuals found so far in the population (this is the elitism which is performed in the innermost section of code).

Parent selection and recombination are implemented in a single step. $\mu/2$ couples (or pairs) are selected by means of μ tournaments. μ is the population size and it is an input parameter. k-size tournament with replacement is used (k is also an input parameter). In tournament selection, k members are randomly chosen from the population and the most

Code 2. AG procedure to solve LC-CSPs

```
AG(input_sequence, μ, k, p, n)
BEGIN
        SET population[0] = input_sequence
        Randomly INITIALIZE the rest of the population μ
        EVALUATE each individual
        SET bests = n individuals with best fitness
        SET n_generations = 0
        REPEAT UNTIL (termination criterion satisfied)
                SELECT μ/2 couples using k size tournaments
                Perform an ORDERED RECOMBINATION of the μ/2 couples
                Offspring SWAP MUTATION with probability p
                ELIMINATE DUPLICATES
                survivors SELECTION for the next generation
                        PERFORM a generational replacement
                        FOR-EACH i in bests
                                IF fitness(i) > fitness(best offspring)
                                        REPLACE a random offspring with i
                        END FOR-EACH
        END REPEAT
END
```

fitting individual wins the tournament and is selected for mating. The process is then repeated to select a new mate. Tournament selection is an easier method to implement and allows a fair degree of control of the selective pressure by modifying the k parameter. It is also commonly preferred to fitness proportional or ranking selection because it does not entail any of the probability bias that they incorporate. The two members of each couple must be different to ensure genetic variation in the offspring. Otherwise, the selection procedure is repeated until they differ. Finally each pair is recombined by means of order crossover, producing μ new individuals. Each pair produces two individuals, so with $\mu/2$ we will obtain μ to keep the population size constant.

The algorithm receives an initial sequence I as an input. This input is used to initialize the first individual. All other individuals are initialized randomly by permuting I. Agent processing stops when a fitness evaluation of an individual returns zero, or when a maximum number of iterations is reached. The number of iterations is also defined as an input parameter (m). To avoid genetic drift (quick convergence to the same or a very similar individual for all the population) a duplicate elimination policy has also been introduced. Just after the recombination and mutation processes each individual is compared with the previous elements in the population. If the genotype is equal to any of these predecessors a swap mutation is enforced until it differs. Duplicate elimination requires full knowledge of the population.

Contrasting with PSO, there is no consensus among researchers concerning the appropriate values to set GA parameters. Despite that, there is agreement that the issue of setting the values of various parameters of an evolutionary algorithm is crucial for good performance [17]. Different strategies for parameter tuning and control have been presented: namely deterministic, adaptive and self-adaptive. Deterministic parameter tuning involves setting every parameter before running the algorithm. It is the simplest strategy but it has some disadvantages: Firstly, the number of tests required increases exponentially as the number of parameters and the values for each parameter increase; and secondly, each configuration is problem-dependent, so parameters must be independently tuned for each problem. Nonetheless, most recent research claims that the use of algorithms to perform the process may be also the best approach [23]. But this is a novel research direction in which more work is required, and these authors also point at the necessity to develop a toolbox of these algorithms. An adaptive strategy requires that parameter values change during the execution, while self-adaptive strategies require that parameters evolve simultaneously with the population and the solution. Both can offer better performance [45], but they increase implementation effort because it is necessary to analyze and assess the parameters' development in order to determine the best strategy. So we decided to try parameter tuning because it offers the best trade-off between performance and development cost [25], but we had to devise a method to reduce the number of required tests. Our approach aims to produce a twofold benefit: First, incorporate a systematic approach to tune GA parameters; and second to try to identify common patterns, and even feasible configurations, concerning parameter tuning for LC-CSPs. These issues will be discussed in the next section.

5. Results.

The algorithms described in the previous section were implemented in C# following the object oriented paradigm. In this section, we describe the experiments that were carried out along with their outcomes. The major part of the work was devoted to parameter setting and optimizer tuning in order to obtain the best performance. All tests were run on a computer with a Pentium 4 2GHz processor and 2GBs of memory.

5.1. PSO experimentation.

Four different configurations are established to test the two new modifications over the original permut-PSO algorithm introduced in Section 4.2.1. These configurations are:

- Configuration 0. All permutations are performed from *gbest*. Comparison for changing particle *pbest* and *gbest* values is set strictly less (<). These are the original settings.
- Configuration 1. All permutations are performed from *gbest*. Comparison for changing particle *pbest* and *gbest* values is set to less than or equal (<=).
- Configuration 2. Permutation of the particle position is randomly selected from *gbest* or from *pbest*. Comparison set to strictly less than (<).
- Configuration 3. Permutations from *gbest*/*pbest*. Comparison set to less than or equal (<=).

Each configuration is then inputted with the 100 different test cases described in Section 4.1, and data results from each execution are collected for statistical analysis. Table 1 presents descriptive statistics of the total number of calls to the fitness function required by each configuration over the 100 tests, and Figure 1 presents the results graphically.

Table 1. Descriptive statistics of number of calls to the fitness function on the 100 test cases of each configuration of the PSO optimizer

Configuration	Mean	Std Error	Std Dev	Minimum	Median	Maximum
0	641.9	23.7	237.0	237	607.5	1762
1	645.6	29.4	293.9	304	564.0	2118
2	1008.6	49.2	491.7	315	932.0	3517
3	975.4	47.1	471.7	289	908.0	2629

The results suggest that configurations 0 and 1 outperform configurations 2 and 3. Thus, the second proposed modification, namely, to permute particles position randomly from *pbest* or *gbest*, does not improve performance but also seems to degrade it considerably. Configurations 0 and 1 offer similar performance on all observed measures although configuration 0 (original settings) seems to be slightly better. This suggests that the second proposed modification, namely, to set the comparison to change *pbest* and *gbest* values to less than or equal instead of strictly less than, does not yield any improvement

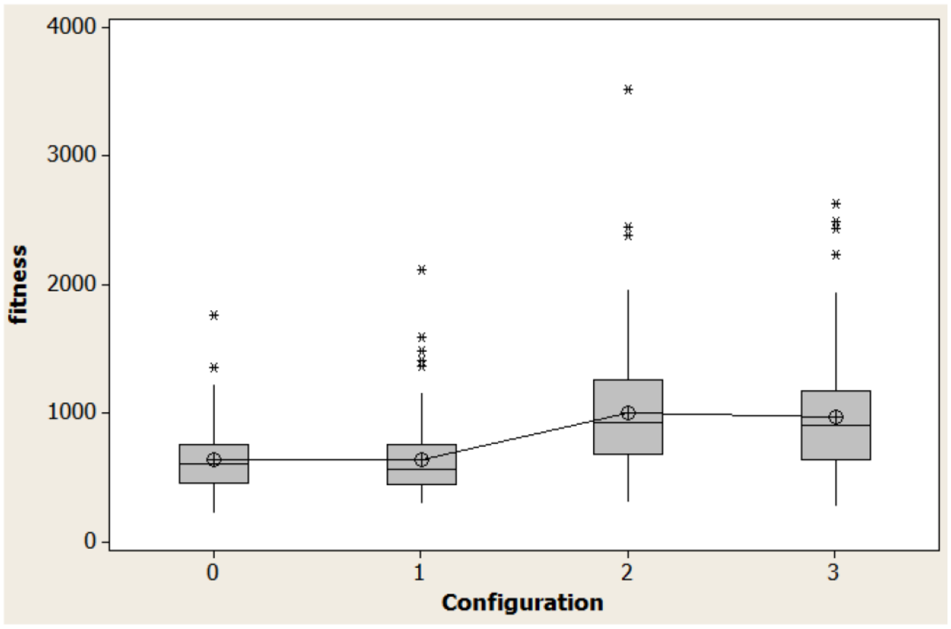

FIGURE 1. Boxplot of fitness of each configuration

in terms of performance. Data transformation is required to confirm these results statistically because gathered data is not normally distributed, but rather it seems to follow a lognormal distribution (Figure 2). This can be explained partially by the long tail (higher values in the number of calls to the fitness function) which may be representative of the most difficult instances of the random problems giving information about the behavior of the problem solvers in the worst case. In this worst case, configuration 0 also seems to outperform all others. A logarithmic transformation (base-10) is then applied and normality tests are also performed. Kolmogorov-Smirnov tests are used to prove normality. Returned p-values are greater than 0.150 for configurations 0, 2 and 3. Unfortunately statistical significance is not so solid for configuration 1 because the p-value returned by the normality test was 0.074. A p-value less than or equal to 0.05 is required for statistical significance (95% confidence interval). Nonetheless, this value also permits the rejection of the null hypothesis and accepts that data follows a normal distribution with a 90% confidence interval. Standard statistical methods can then be applied to the transformed data in order to check the aforementioned hypotheses.

After data transformation and normality checking, a one-way ANOVA can be run to confirm the differences and similitudes in terms of the performance of the different configurations. There is a statistical significance of the performance observed for the different configurations ($p = 0.00$ and $F = 39.98$), although the model does not fit the data particularly well ($R^2 = 19.01\%$). In order to show such differences graphically, Figure 3 presents the confidence intervals of fitness for each configuration (95% CI). Additionally, six paired t-tests are run to compare each possible pair of configurations. The resulting *p*-values (presented in Table 2) demonstrate that there is enough statistical significance to state that configurations 0 and 1 outperform configurations 2 and 3, and that there is no statistical significance to conclude which of each of these pairs is the best choice, i.e., it cannot be concluded that configuration 0 outperforms configuration 1 or that configuration 2 outperforms configuration 3.

To evaluate the potential benefit yielded by configuration 0, an additional statistical analysis is performed. Hsu's MCB (Multiple Comparisons with Best) test is used to compare each mean with the best (i.e., the smallest because it is a minimization problem)

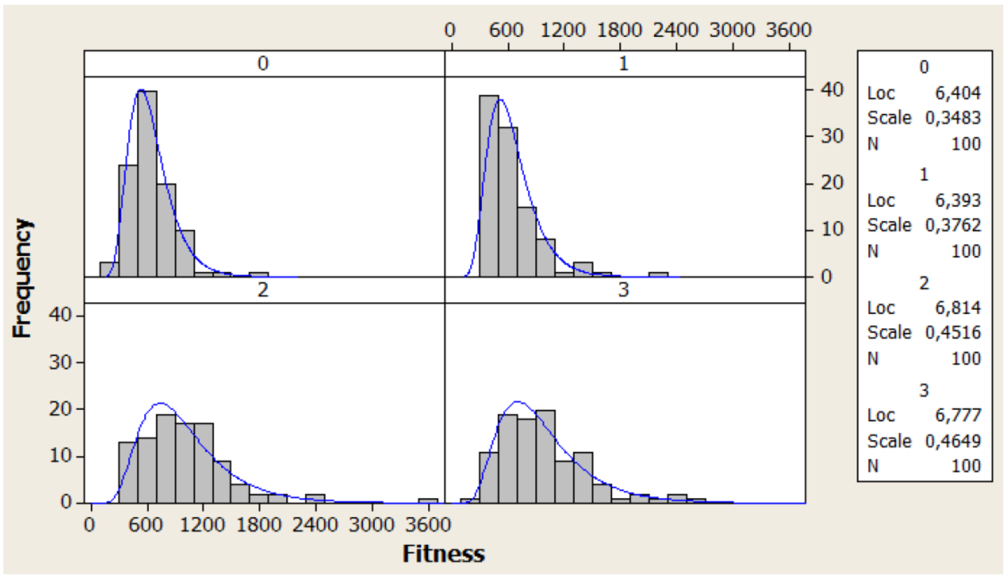

FIGURE 2. Histogram of fitness for each configuration with lognormal fit

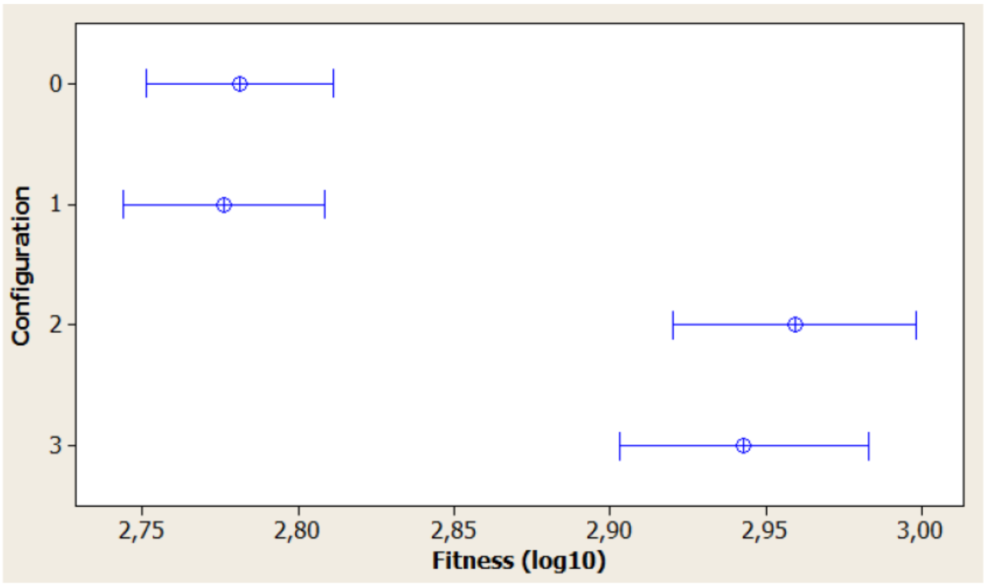

FIGURE 3. Confidence intervals of fitness for each configuration (CI = 95% of the mean)

TABLE 2. p-values returned by paired t-tests between each pair of configurations

	Configuration 1	Configuration 2	Configuration 3
Configuration 0	.839	.000	.000
Configuration 1	–	.000	.000
Configuration 2	–	–	.567

of the other means. Hsu's MCB test returned the intervals for level's means minus the smallest of other means (Table 3). To make the results more informative the inverse transformation is applied, and table 3 presents the number of calls to the fitness function. There is no evidence as to whether configuration 2 or 3 is the best because the lower interval endpoints are 0. The potential advantage and disadvantage of any configuration can be further analyzed with this information. Configuration 0 is the best and it is as much as 117.2 better than configuration 1, but it may be as much as 109.9 worse than the best mean of configuration 1. There is therefore no significant difference between the best and the worst cases for both configurations. Such differences are likely to be inconsequential from a practical point of view, and can lead us to determine that both configurations exhibit the same performance.

Table 3. Intervals of Hsu's MCB test of the four configurations

Configuration	Lower	Center	Upper
0	−117.2	−3.7	109.9
1	−109.9	3.7	117.2
2	0.0	366.6	480.2
3	0.0	333.5	447.0

We can conclude that the canonical version of the PSO algorithm described in current literature to deal with permutation CSPs can also satisfactorily solve LC-permut-CSPs. Existing recommendations for parameter setting also seem to suffice for solving all presented instances. Modifications of the original algorithm are apparently not necessary because the two adjustments presented and tested do not produce any benefit.

5.2. **Genetic algorithm experimentation.** During the experimentation phase with the genetic algorithm, our work is mainly related to parameter tuning. The standard genetic algorithm presented above has four input parameters that may influence performance. Our aim is to offer a systematic approach for parameter tuning that can be used to establish good parameter configurations to solve LC-permut-CSPs. The first steps then involve the selection of the test values for each parameter, and the establishment of a tuning strategy. Test values are selected to cover the widest possible range. For the population size (μ), populations of 10, 20, 50 and 100 individuals are tested. Population sizes represent the number of the individuals that are part of the population in each generation (iteration). As for the mutation probability (p), values of .0, .1, .5 and 1 are selected for testing. These represents the probability that two random positions of each newly created individual are a swap. We select the lower and upper limits, a low value and the median value. For the other two parameters, some of their values are population-based, which means that the final value in these cases is determined by the population size and they will change if μ changes. For the tournament size (k), the values selected for experimental testing are 2, 3, $\mu/3$, $\mu/2$ and $2\mu/3$. The tournament size represents the number of individuals that are randomly picked to take part in the tournament. The best individual is then selected for crossover. It is well known that this parameter has a decisive influence on selective pressure. Our decision is to select the values that are on the boundary and some values uniformly distributed in-between. A value of 1 was excluded because it supposes a random selection which contradicts any rationale of a fitness-based selection. On the other boundary, a value equal to the population size results in the two fittest individuals always being selected and precludes all other individuals from participating in the mating process. This will critically affect the variability of the population and that was the reason for excluding values larger than $2\mu/3$. For the final parameter, elitism size (n), the values are 0, 1, $\mu/3$, $\mu/2$, $2\mu/3$ and $\mu - 1$. Elitism size represents the number of individuals that are kept track of for their inclusion in subsequent generations. The best individuals found so far are introduced after the replacement if most fitted individuals of the current generation do not improve current best values. When boundary values are 0, there is no elitism at all, and when $\mu - 1$ all individuals but one can be transferred to the next generation. Thus, in any case one individual of the new generation is always introduced into the following one if it returns a good fitness. All other values for this parameter are uniformly distributed between these boundaries. This selection of values for each parameter may seem to be arbitrary. Some influences may be traced back to existing literature, but in any case our intentions are to test different values to assess their performance and to cover the widest possible range. We try to select the upper and lower bounds in each case and some median values in order to find, not the optimum values, but the optimal areas in which those values can be found. From this perspective, our selection is always open to considering new values for experimental testing. These assumptions also influenced our tuning strategy.

An exhaustive test of all possible configurations demands 480 different settings. In order to reduce this number to a more reasonable number, but that is also susceptible to thorough statistical analysis, we devise an iterative approach which is a kind of hill climbing on the parametric search space. Firstly, an arbitrary setting, called the 'pivot' setting, of values is established. Then in a first iterative process, the values of every parameter are kept constant with the exception of one: the parameter that is going to be tested. All the values of this parameter are subsequently tested and data about the runs is collected. Then the process is repeated for the remaining parameters. The values of every parameter that is not being tested remain constant, keeping the value defined in the pivot set. Finally, statistical analysis was performed to determine the region in which the optimal values for each parameter can be found. At this point the parameter values and the pivot set can be modified and the whole process is then repeated. In this way we will reduce the initial number of configurations to be tested from 480 to 16. This may not find the perfect configuration but can direct us towards it, offering a good understanding about the parameters' influence in the final outcome. We think that this may be a suitable method because: (1) we use an arbitrary selection as a departure point, (2) we explore a wide range of parameter values to cover as much of the parameter space as possible, and (3) we use a pseudo-evolutive process to iteratively improve the parameter setting. We thus think that it is also a suitable method to be applied in parameter setting for practical problems because it can offer a good balance between performance and implementation effort.

The pivot setting is determined taking the central value for each parameter. When the number of values to be tested is even, the decision is arbitrary. Thus the pivot setting is $\mu = 20$, $k = \mu/3$, $p = 0.1$, $n = \mu/2$. Each tested configuration is input with the 100 test cases while performance data is gathered for statistical analysis. Table 4 presents the descriptive statistics of the number of calls to the fitness function for each of the 16 settings that are tested, grouped by parameters. Results suggest that the selection of the population size, tournament size and elitism size can critically influence performance. Furthermore, some sets of values seem to yield much better results than others, and some particular values seem to be completely inadequate. On the other hand, the value for the mutation probability does not seem to be so important. To confirm such intuitions a statistical analysis is subsequently performed.

Each parameter in the study can be considered as a variable and a General Linear Model (GLM) can be run to determine which variables influence the final outcome. But previous data transformation is required because data does not follow a normal distribution. Rather data seems to fit better to a lognormal distribution. Lognormal distributions are characterized by a long tail on the right side. That means that a number of instances of the problem are more difficult to solve. This phenomenon has been explained in studies about phase transition and is due to the fact that in the 'easy' region there are some hard problems which can be randomly encountered [30]. A logarithmic transformation

Table 4. Descriptive statistics of number of calls to the fitness function of each parameter on the 100 test cases

Value	Mean	Std Error	Std Dev	Minimum	Median	Maximum
Parameter: Population size (μ)						
10	1323.8	52.9	529.4	550	1190	3360
20 (pivot)	1005.0	45.1	451.4	300	860	2840
50	1113.5	53.5	534.7	450	950	3050
100	1297.0	49.6	495.7	600	1200	4600
Parameter: Tournament Size (k)						
2	3150.0	119.0	1187.0	940	2990	7640
3	1921.6	64.4	644.3	540	1830	4100
$\mu/3$ (pivot)	1005.0	45.1	451.4	300	860	2840
$\mu/2$	1047.2	58.5	585.5	340	920	3480
$2\mu/3$	985.6	52.7	527.0	300	850	2920
Parameter: Probability (p)						
0	1065.0	43.4	433.7	420	970	2860
.1 (pivot)	1005.5	45.1	451.1	300	860	2840
.5	1002.6	38.0	380.3	360	950	2140
1	1020.6	33.1	331.3	500	970	2120
Parameter: Elitism Size (n)						
0	1293.4	64.6	646.4	380	1120	3660
1	1047.4	50.2	502.0	320	960	3720
$\mu/3$	1067.0	54.4	544.8	440	940	3720
$\mu/2$ (pivot)	1005.0	45.1	451.4	300	860	2840
$2\mu/3$	1038.4	50.5	505.1	340	940	3080
$\mu - 1$	1034.4	47.3	473.1	360	890	2820

Table 5. Results returned by a general linear model for each parameter ($R^2 = 36.32\%$)

Source	F	p
Population size (μ)	13.05	.000
Tournament size (k)	180.28	.000
Probability (p)	.61	.610
Elitism size (n)	4.30	.001

(base-10) is then applied and normality tests are run. Kolmogorov-Smirnov tests are also used to prove normality. All 16 tests are successful with a confidence interval of 95%. The GLM can then be run, and results are summarized in Table 5. There is significant evidence of the effects of three parameters (μ, k and n) on the observed performance, while the other parameter (p) does not influence performance in a significant manner.

Confidence intervals for each parameter are presented in Figures 4-7. They offer a graphical representation of the possible influence of each value on the final performance. Individual tests for each parameter are subsequently run to support the graphical results. They are presented in the remainder of this section.

We firstly evaluate the differences for the population size. A one way ANOVA with Hsu's MCB test is run to compare each mean with the best of the other means. This test returned the intervals for level means minus smallest of other means. The results are

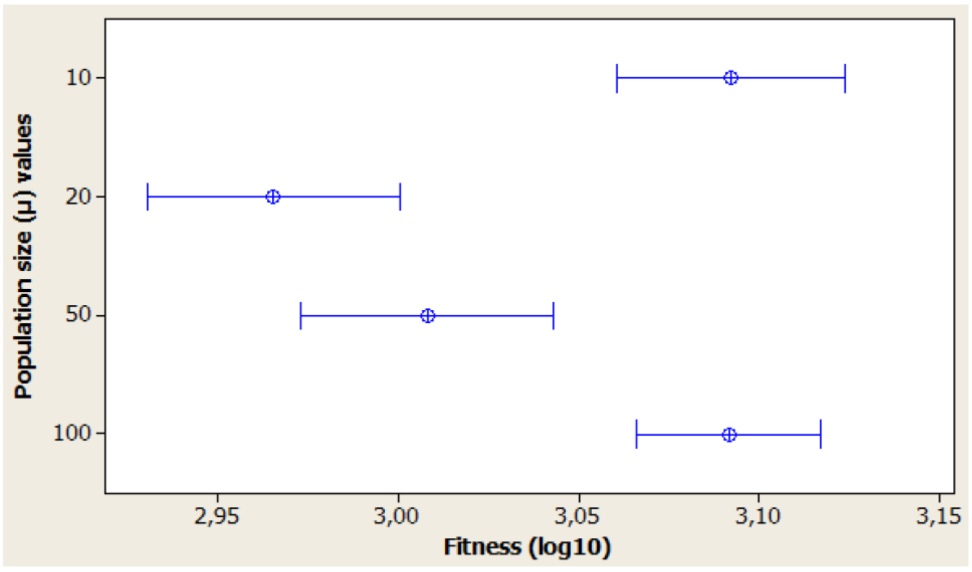

FIGURE 4. Confidence intervals of fitness for each value of the population size (μ) parameter (CI = 95% of the mean)

TABLE 6. Intervals of Hsu's MCB test for the population size (μ)

Value	Lower	Center	Upper
10	.0	318.8	465.7
20	−255.4	−108.5	38.5
50	−38.5	108.5	255.4
100	.0	292.0	438.9

TABLE 7. Intervals of Hsu's MCB test for the tournament size (k)

Value	Lower	Center	Upper
2	.0	2164.8	2387.2
3	.0	936.0	1158.4
$\mu/3$	−203.0	19.4	241.8
$\mu/2$	−160.8	61.6	284.0
$2\mu/3$	−241.8	−19.4	203.0

presented in Table 6. Populations of 20 and 50 individuals are statistically better because both intervals include 0. Although there is no statistical significance to conclude that any of one is better than the other, we can also conclude that $\mu = 20$ would require 255.4 less calls to the fitness function than $\mu = 50$ in the best case and just 38.5 more calls in the worst case. It is then important to select a correct value for the population size and $\mu = 20$ offers the best performance for this problem.

As for the tournament size (k), confidence intervals are presented in Figure 5 and the results of the Hsu's MCB test are summarized in Table 7. Low values for the tournament size are statistically worse and their differences in terms of performance with larger values are important. If we focus on values larger than $\mu/3$ it can be observed that there is not enough statistical significance concerning performance, but the larger tested value ($2\mu/3$) seems to perform slightly better. If it is to be the best setting it would be at best 241.8 calls better than its closest competitor and at worst 203 calls worse. Differences between the largest three settings are thus below 8.23% over the mean.

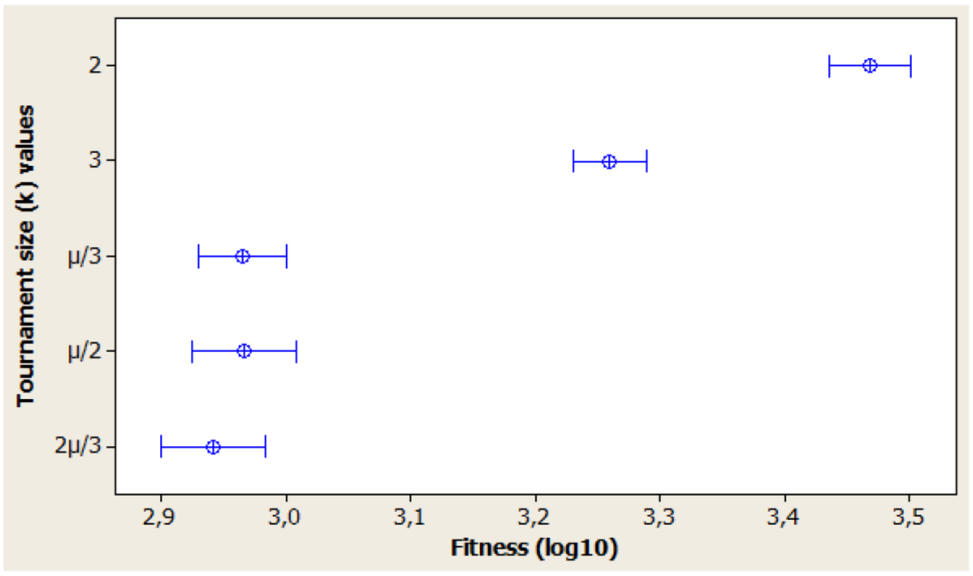

FIGURE 5. Confidence intervals of fitness for each value of the tournament size (k) parameter (CI = 95% of the mean)

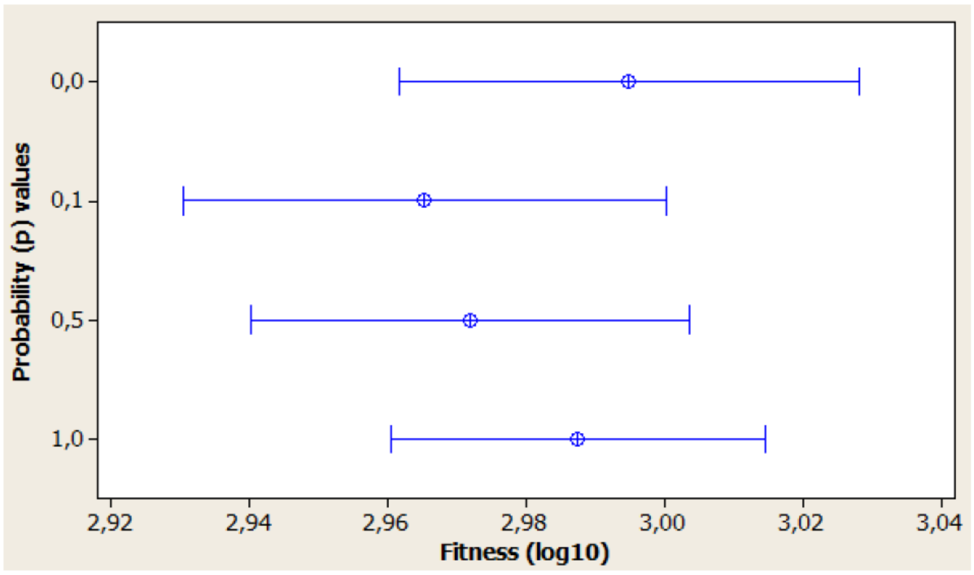

FIGURE 6. Confidence intervals of fitness for each value of the probability (p) parameter (CI = 95% of the mean)

Figure 6 presents the confidence intervals for the mutation probability (p) and Table 8 presents the results of the Hsu's MCB test. For this parameter there is not enough statistical evidence to determine whether any setting is better or worse than any other. An analysis of the data reveals that .5 is the best value, but performance yielded is below 6.5% over the mean of the worst value (0), and below 2% if any of the largest values are selected.

Finally, Figure 7 and Table 9 present the results for the elitism size (n) parameter. When elitism is enabled ($n > 0$) the performance is better, which is supported by statistical evidence. Besides that, $\mu/2$ returns the best performance but that would mean just 91 fewer calls (9.05% over the mean) to the fitness function in the best case.

We can conclude that a simple version of a GA aligned with common trends in literature can be successfully employed to solve LC-permut-CSPs. The main effort has been devoted

TABLE 8. Intervals of Hsu's MCB test for the mutation probability (p)

Value	Lower	Center	Upper
.0	−54.8	62.4	179.6
.1	−114.8	2.4	119.6
.5	−119.6	−2.4	114.8
1	−99.2	18.0	135.2

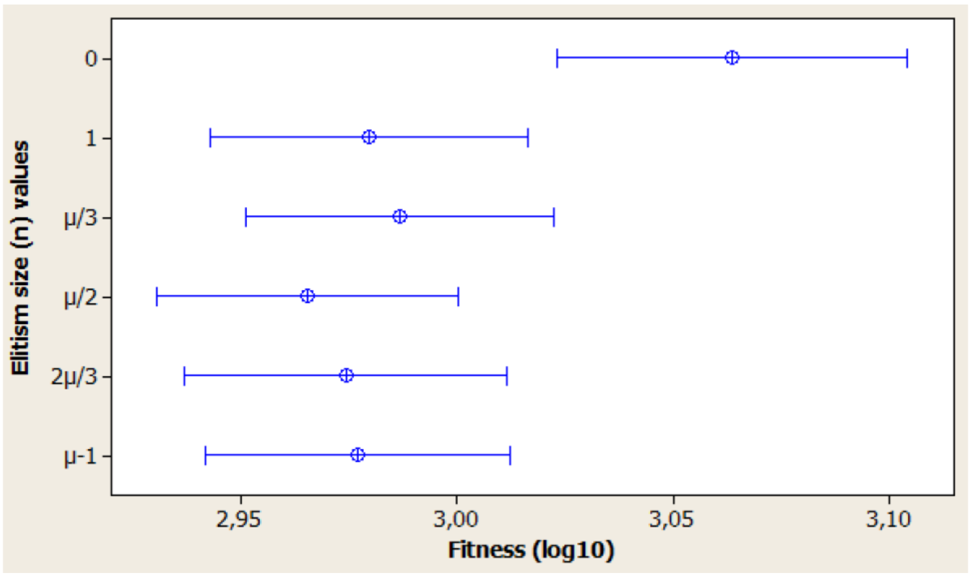

FIGURE 7. Confidence intervals of fitness for each value of the elitism size (n) parameter (CI = 95% of the mean)

TABLE 9. Intervals of Hsu's MCB test for the elitism size (n)

Value	Lower	Center	Upper
0	.0	288.4	454.0
1	−123.2	42.4	208.0
$\mu/3$	−103.6	62.0	227.6
$\mu/2$	−195.0	−29.4	136.2
$2\mu/3$	−132.2	33.4	199.0
$\mu - 1$	−136.2	29.4	195.0

to parameter tuning. In this respect, we have seen that most parameters are important and we can summarize that it is more important to avoid unsuitable values than to strive for setting the best one. Bad decisions will critically affect performance while fine-tuning yields limited improvements. Focusing on each parameter individually, we can summarize our findings as follows: Population size (μ) seems to be the most important value and the only one that it would be recommendable to fine tune. We surmise that the best value may depend on the problem type and size. For the experimental problems under study, the best value was 20. Low values for the tournament size (k) should be avoided. Setting k to values larger than $\mu/3$ is recommendable but no substantial improvement is observed between the different values greater than this. Mutation probability (p) does not critically influence performance. A value larger than .5 has a limited effect on performance that can make it preferable. Elitism improves performance when it is enabled, so it is recommended to set elitism size parameter (n) to a value larger than 0. No relevant differences are observed for other values of this parameter.

5.3. **Comparative analysis.** Both approaches are finally compared to test their relative performance to solve LC-permut-CSPs. The best configurations found for each algorithm are used for the comparative analysis. For the PSO the canonical version was used. As for the GA, the optimal parameter settings determined by previous experimentation are employed ($\mu = 20$, $k = 2\mu/3$, $p = .1$, $n = \mu/2$). An initial overview of descriptive statistics of both algorithms seems to shows that the PSO approach outperforms the GA. Figure 8 presents a boxplot that depicts the difference visually. This is confirmed with an ANOVA test (F = 38.28, p = .000, $R^2 = 16.2$) and the confidence intervals presented in Figure 9. We can then conclude that a canonical version of the PSO is a better option to solve random LC-permut-CSPs. PSO also has less parameters and tuning is actually not required if we follow the recommendations available in the related literature. On the other hand, GA has several parameters in which decisions need to be made and different tests need to be performed. Considering that the kind of experimental random LC-CSPs that we have employed here bears an important resemblances with many real world optimization problems, this study suggests that such problems should firstly be approached using a PSO algorithm rather than a GA. This would result in better performance in terms of efficiency as well as in terms of development costs.

5.4. **Practical example and comparison with other works.** So far we have presented our framework for testing and tuning evolutionary algorithms used to solve LC-permut-CSPs, but we have no presented any particular instance. In this subsection we present an example of the execution of the solver for a hypothetical task scheduling problem and we also compare this work with a practical example from the literature. Figure 10 presents the input and output of a generic task scheduling problem. The input represents a set of random tasks generated accordingly to the test case described in Section 4.1 that contains 24 tasks and 35 randomly generated binary constraints. It can represent a set of tasks to be performed on a hypothetical machine in a factory. On the output set, pre-tasks are then presented at the beginning of the output sequence. They are followed

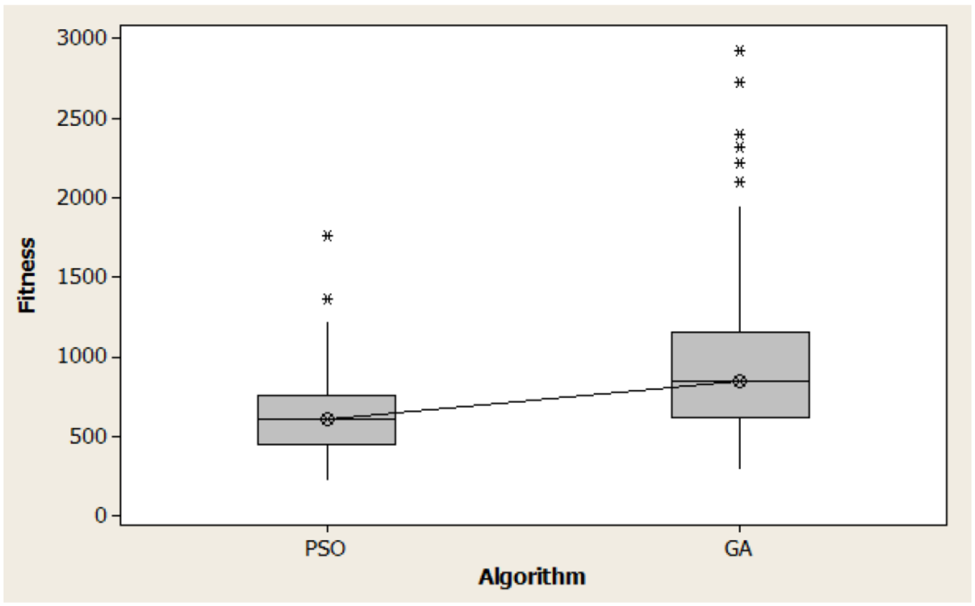

FIGURE 8. Boxplot of fitness of the PSO and the GA

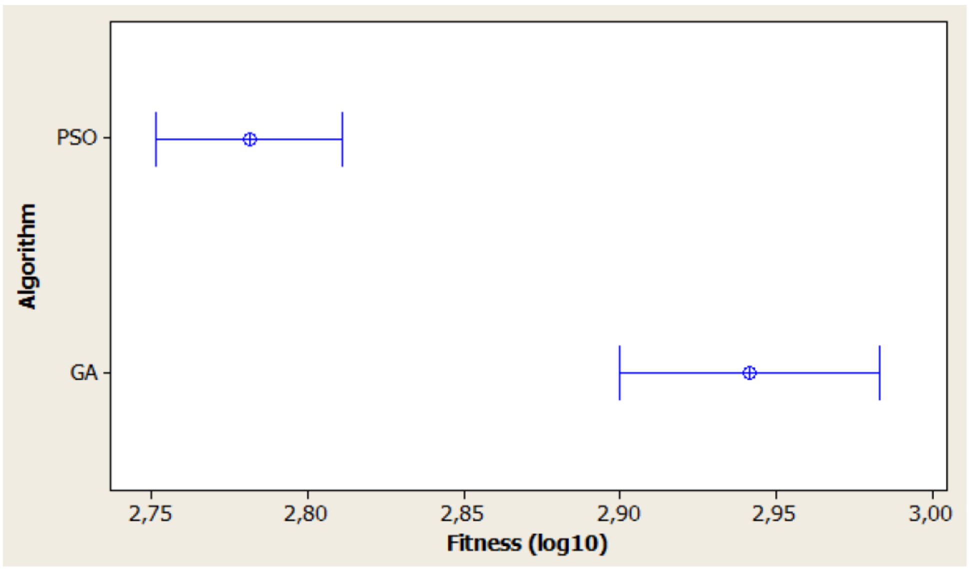

FIGURE 9. Confidence intervals of fitness for each algorithm (CI = 95% of the mean)

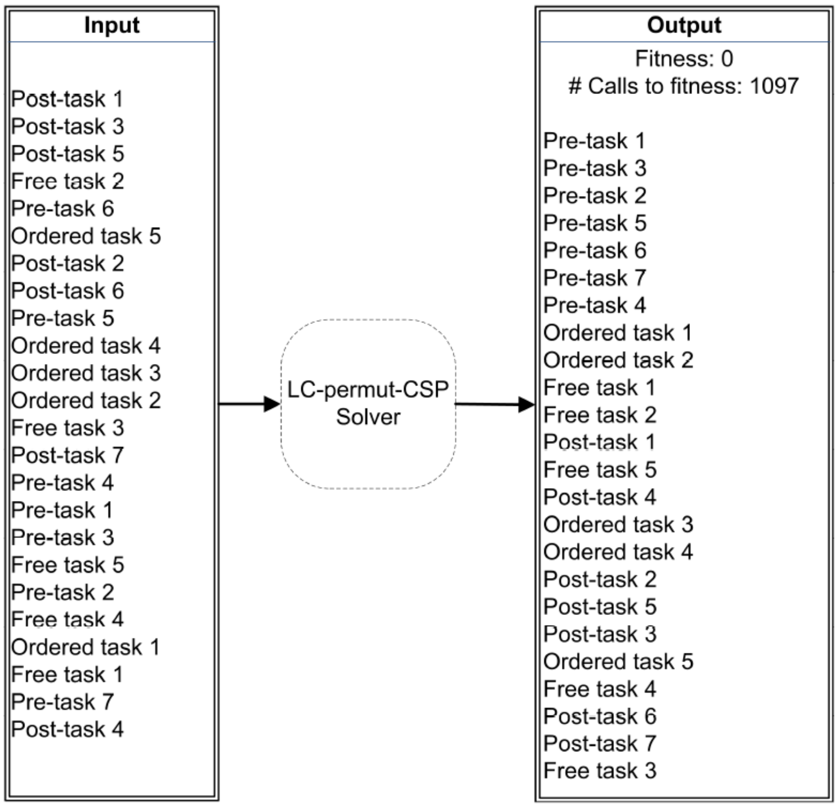

FIGURE 10. Example of a task scheduling problem

by the ordered tasks, post-tasks and free tasks. It can be observed that the ordered tasks are performed in an appropriate sequence and that the rest of the tasks are inserted in the sequence respecting every constraint. The output is complemented with the final value of the fitness function 0 (all constraints have been satisfied) and the number of calls to the fitness function required to obtain the final result.

De-Marcos et al. [46] present a practical example that uses genetic algorithms to sequence learning units. The authors present a real world application that comprises a set of courses (in four different categories) that must be arranged in a particular order prior to their delivery to the student. They use a genetic algorithm that successfully obtains a solution for every instance presented. This application could benefit from our framework since the problem can be modeled as a LC-permut-CSP, and then it would be susceptible to systematic tuning and testing. As our work demonstrates, a PSO approach will likely produce better results since a courseware engineering solver can be modeled as LC-permut-CSPs. Our work can help researchers and engineers select the technique and algorithm that will (potentially) produce the best result for this class of problems and also liberate them from the necessity of tuning algorithms.

6. **Conclusions and Future Work.**

The main features of LC-permut-CSPs, as found in real world applications, have been abstracted to a generic test case that has been used as the basis for the systematic analysis of two different problem solvers. The canonical version of a PSO optimizer has been analyzed first. Two different modifications have been introduced. They are designed to improve performance in the discrete landscape in which the problem solver has to work. A statistical analysis has been carried out to assess and compare the performance observed for each possible configuration. Results show that the canonical version of the PSO performs well when solving the test case, and that no modifications over the original version are required. The second problem solver is a standard GA. In this case parameters exist that require tuning, and our efforts have focused on performing another statistical analysis to assess the influence of each parameter on the observed performance. Results have shown that all parameters except one can critically affect the behavior of the problem solver. Guidance can be offered in terms of the values that should be avoided instead of specific values to be set. When an appropriate setting is chosen, we have observed that the GA can also deal with all the LC-permut-CSP instances that have been tested. Finally, a comparative analysis has been carried out. Statistical evidence has been found to assert that the PSO agent outperforms the GA approach.

This work has a special focus on the trade-off between performance and development costs. Our idea is to establish a common framework to test and compare canonical versions of different methods. The authors believe that, for example, it would be possible to find a GA that offers a performance similar to that of the PSO, but the cost of doing this it would probably be prohibitive from the point of view of practitioners. From that perspective, we have focused on the canonical versions of both methods and presented a systematic approach to build, design, test and compare them, also trying to avoid, or at least relieve, the costly and complex stages which developers will incur to design solutions for LC-permut-CSPs. Our findings suggest that both approaches work but that a canonical version of the PSO should be preferred.

Our work focuses on simple binary CSPs as they are easy to model and analyze. This mathematical reduction is deemed necessary since the complexity for defining and analyzing CSPs that consider constraints of higher degrees can be very high. But we also want to note that many of the constraints that can be found in the real world are not binary but rather of an n-ary nature (i.e., involving three or more variables). Thus the framework proposed here would be of little practical use. A mathematical apparatus can be developed and used to deal with this kind of constraint, but the authors think that a more reasonable approach would be to develop the necessary methods that enable the transformation of n-ary CSPs into binary CSPs. Also, the test case does not consider the existence on any specific weight associated with every variable to prioritize them.

Such weights appear in different practical problems needing to deal with cost, time or the priority on their optimization. It should also be noted that the algorithms tested here are stochastic (indeed, many metaheuristic methods are). Stochastic methods produce different solutions for the same problem if they are run different times. There is evidence that in practical applications where the user manipulates the algorithm output this can produce confusion on the part of the user since computers are expected to be deterministic. LC-permut-CSPs aggravate this problem because many solutions exist for any instance. This limitation shall be further studied to improve the efficiency and usability of our approach.

The obvious way to extend our work will be to include more methods in the toolset. Other evolutionary methods like artificial ants have also been used to solve CSPs [22]. Additional bio-inspired approaches that have not been devised or adapted to solve LC-permut-CSPs, like artificial bee colonies [47], can also be considered. Methods based on social metaphors have also been used for optimization purposes. The parliamentary metaphor has already been used to solve CSPs [48] and related methods can also be extended to solve them. The latter may include, just to mention a few, those that are inspired from social behavior (like [49]) or cultural algorithms, which have already been used to solve combinatorial optimization problems ([50] presents a recent example).

REFERENCES

[1] E. Tsang, *Foundations of Constraint Satisfaction*, Academic Press, London, 1993.

[2] A. Banks, J. Vincent and C. Anyakoha, A review of particle swarm optimization. Part II: Hybridisation, combinatorial, multicriteria and constrained optimization, and indicative applications, *Nat. Comput.*, vol.7, no.1, pp.1567-7818, 2008.

[3] L. Davis, *Handbook of Genetic Algorithms*, Thomson Publishing, Washington, USA, 1991.

[4] R. Poli, J. Kennedy, T. Blackwell and A. Freitas, Analysis of the publications on the applications of particle swarm optimisation, *J. Artif. Evol. Applic.*, vol.2008, no.1, pp.1-10, 2008.

[5] A. Yardimci, Soft computing in medicine, *Appl. Soft Comput.*, vol.9, no.3, pp.1029-1043, 2009.

[6] A. Bouchachia, R. Mittermeir, P. Sielecky, S. Stafiej and M. Zieminski, Nature-inspired techniques for conformance testing of object-oriented software, *Appl. Soft Comput.*, vol.10, no.3, pp.730-745, 2010.

[7] T. Mantere and J. T. Alander, Evolutionary software engineering, a review, *Appl. Soft Comput.*, vol.5, no.3, pp.315-331, 2005.

[8] D.-F. Shiau, A hybrid particle swarm optimization for a university course scheduling problem with flexible preferences, *Expert Syst. Appl.*, vol.38, no.1, pp.235-248, 2011.

[9] L. de-Marcos, J. J. Martínez, J. A. Gutiérrez, R. Barchino and J. M. Gutiérrez, A new sequencing method in web-based education, *Proc. of the IEEE Congress on Evolutionary Computation*, Trondheim, Norway, pp.3219-3225, 2009.

[10] V. Oduguwa, A. Tiwari and R. Roy, Evolutionary computing in manufacturing industry: An overview of recent applications, *Appl. Soft Comput.*, vol.5, no.3, pp.281-299, 2005.

[11] J. Robinson and Y. Rahmat-Samii, Particle swarm optimization in electromagnetics, *IEEE Trans. on Antennas and Propagation*, vol.52, no.2, pp.397-407, 2004.

[12] J. H. Holland, *Adaptation in Natural and Artificial Systems*, The University of Michigan Press, Michigan, USA, 1975.

[13] R. Eberhart and J. Kennedy, A new optimizer using particle swarm theory, *Proc. of the 6th International Symposium on Micro Machine and Human Science*, Nagoya, Japan, 1995.

[14] J. Kennedy and R. Eberhart, Particle swarm optimization, *Proc. of the IEEE International Conference on Neural Networks*, Perth, WA, Australia, 1995.

[15] T. Hogg, Refining the phase transition in combinatorial search, *Artif. Intell.*, vol.81, no.1-2, pp.127-154, 1996.

[16] P. Prosser, An empirical study of phase transitions in binary constraint satisfaction problems, *Artif. Intell.*, vol.81, pp.81-109, 1996.

[17] A. E. Eiben, R. Hinterding and Z. Michalewicz, Parameter control in evolutionary algorithms, *IEEE Trans. Evol. Comput.*, vol.3, no.2, pp.124-141, 1999.

[18] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, Readomg, USA, 1989.

[19] J. Kennedy and R. C. Eberhart, A discrete binary version of the particle swarm algorithm, *Proc. of IEEE International Conference on Systems, Man, and Cybernetics*, Orlando, FL, 1997.

[20] X. Hu, R. C. Eberhart and Y. Shi, Swarm intelligence for permutation optimization: A case study of n-queens problem, *Proc, of the IEEE Swarm Intelligence Symposium*, Indianapolis, USA, 2003.

[21] L. Schoofs and B. Naudts, Ant colonies are good at solving constraint satisfaction problems, *Proc. of the Congress on Evolutionary Computation*, La Jolla, CA, 2000.

[22] C. Solnon, Ants can solve constraint satisfaction problems, *IEEE Trans. Evol. Comput.*, vol.6, no.4, pp.347-356, 2002.

[23] S. K. Smit and A. E. Eiben, Comparing parameter tuning methods for evolutionary algorithms, *IEEE Congress on Evolutionary Computation*, 2009.

[24] R. Tan, S. Wang, Y. Jiang, K. Ishida and M. G. Fujie, Motion control with parameter optimization by genetic algorithm, *ICIC Express Letters*, vol.5, no.8(B), pp.2779-2784, 2011.

[25] V. Nannen, S. Smit and A. Eiben, Costs and benefits of tuning parameters of evolutionary algorithms, in *Parallel Problem Solving from Nature – PPSN X*, G. Rudolph et al. (eds.), Springer, 2008.

[26] B. Jarboui, N. Damak, P. Siarry and A. Rebai, A combinatorial particle swarm optimization for solving multi-mode resource-constrained project scheduling problems, *Appl. Math. Comput.*, vol.195, no.1, pp.299-308, 2008.

[27] D. Merkle, M. Middendorf and H. Schmeck, Ant colony optimization for resource-constrained project scheduling, *IEEE Trans. Evol. Comput.*, vol.6, no.4, pp.333-346, 2002.

[28] C.-P. Chu, Y.-C. Chang and C.-C. Tsai, PC^{2}PSO: Personalized e-course composition based on particle swarm optimization, *Appl. Intell.*, vol.34, no.1, pp.141-154, 2009.

[29] L. de-Marcos, R. Barchino, J. J. Martinez and J. A. Gutierrez, A new method for domain independent curriculum sequencing: A case study in a web engineering master program, *Int. J. Eng. Educ.*, vol.25, no.4, pp.632-645, 2009.

[30] T. Hogg, B. A. Huberman and C. P. Williams, Phase transitions and the search problem, *Artif. Intell.*, vol.81, pp.1-15, 1996.

[31] M. Mitchell, *An Introduction to Genetic Algorithms (Complex Adaptive Systems)*, The MIT Press, Cambridge, MA, 1998.

[32] R. Mendes, J. Kennedy and J. Neves, The fully informed particle swarm: Simpler, maybe better, *IEEE Trans. Evol. Comput.*, vol.8, no.3, pp.204-210, 2004.

[33] A. Banks, J. Vincent and C. Anyakoha, A review of particle swarm optimization. Part I: background and development, *Nat. Comput.*, vol.6, no.4, pp.467-448, 2007.

[34] Y. Shi and R. Eberhart, Parameter selection in particle swarm optimization, *Proc. of the 7th International Conference on Evolutionary Programming*, San Diego, USA, 1998.

[35] T. Bäck, L. J. Fogel and Z. Michalewicz, *Evolutionary Computation 1: Basic Algorithms and Operators*, Institute of Physics Publishing, Bristol, UK, 2000.

[36] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing (Natural Computing Series)*, Springer, Berlin, Germany, 2003.

[37] G. Syswerda, Schedule optimisation using genetic algorithms, in *Handbook of Genetic Algorithms*, L. Davis (ed.), Thomson Publishing, Washington, USA, 1991.

[38] D. E. Goldberg and R. Lingle, Alleles, loci and the traveling salesman problem, *Proc. of the 1st International Conference on Genetic Algorithms and their Application*, Lawrence Erlbaum, Cambridge, USA, 1985.

[39] L. D. Whitley, *Permutations, Evolutionary Computation 1: Basic Algorithms and Operators*, T. Bäck, L.J. Fogel and Z. Michalewicz (eds.), Institute of Physics Publishing, Bristol, UK, 2000.

[40] I. M. Olivier, D. J. Smith and J. H. Holland, A study of permutation crossover operators on the traveling salesman problem, *Proc. of the 1st International Conference on Genetic Algorithms and Their Application*, Cambridge, USA, 1985.

[41] K. de Jong and J. Sarma, On decentralizing selection algorithms, *Proc. of the 6th international Conference on Genetic Algorithms*, San Francisco, CA, USA, pp.17-23, 1995.

[42] J. E. Baker, Reducing bias and inefficiency in the selection algorithm. *Proc. of the 2nd International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*, pp.14-21, 1987.

[43] T. Bäck, Generalised convergence models for tournament and (μ, λ) selection, *Proc. of the 6th International Conference on Genetic Algorithms*, pp.2-8, 1995.

[44] L. D. Whitley and J. Kauth, Genitor: A different genetic algorithm. *Proc. of the Rocky Mountain Conference on Artificial Intelligence*, pp.118-130, 1988.

[45] A. Eiben, M. Schut and A. de Wilde, Is self-adaptation of selection pressure and population size possible? – A case study, *Parallel Problem Solving from Nature – PPSN IX, LNCS*, pp.900-909, 2006.
[46] L. de-Marcos, J. J. Martinez, J. A. Gutierrez, R. Barchino, J. R. Hilera, S. Otón and J. M. Gutierrez, Genetic algorithms for courseware engineering, *International Journal of Innovative Computing, Information and Control*, vol.7, no.7, pp.3981-4004, 2011.
[47] D. Karaboga and B. Basturk, On the performance of artificial bee colony (ABC) algorithm, *Appl. Soft Comput.*, vol.8, no.1, pp.687-697, 2008.
[48] L. de-Marcos, A. García, E. García, J. J. Martínez, J. A. Gutiérrez, R. Barchino, J. M. Gutiérrez, J. R. Hilera and S. Otón, An adaptation of the parliamentary metaheuristic for permutation constraint satisfaction, *Proc. of the IEEE Congress on Evolutionary Computation*, Barcelona, Spain, pp.834-841, 2010.
[49] T. Ray and K. M. Liew, Society and civilization: An optimization algorithm based on the simulation of social behavior, *IEEE Trans. Evol. Comput.*, vol.7, no.4, pp.386-396, 2003.
[50] C. Soza, R. L. Becerra, M. C. Riff and C. A. C. Coello, Solving timetabling problems using a cultural algorithm, *Appl. Soft Comput.*, vol.11, no.1, pp.337-344, 2011.