Abstract— The design of well-performing heuristic-based algorithms is a tedious and time-consuming task. For this purpose, several approaches to automate the algorithm design have been investigated over the last years. Among them, Genetic Programming-based Hyper-Heuristics (GPHH) are able to design algorithms of varying lengths and permutations of algorithm’s operators. Although the parametrization of generated algorithms is crucial for further adaptation of its behavior to the underlying problems, it has been neglected or only considered to a limited extent in many applications of GPHHs. This work proposes a way to integrate GPHH with parameter configuration. In particular, a Cartesian Genetic Programming Hyper-Heuristic (CGPHH) is developed. The proposed method is applied to the multi-level capacitated production lot-sizing problem, a complex and relevant problem in production planning. Conducted experiments evaluate both its efficacy and the quality of the generated algorithms. Finally, results are compared to the CGPHH without parameter configuration and a human-designed algorithm, indicating the ability of CGPHH-PC to generate algorithms that have the best overall-performance for different cutoff times.

Keywords—Hyper-heuristics, Genetic Programming, Algorithm Generation, Parameter Configuration, Lot-sizing Problem.

I. INTRODUCTION

Heuristic-based methods, such as meta-heuristics, have been successfully applied to solve a multitude of complex problems. To achieve good results for a particular domain, its operators and parameters need to be adapted, and the performance assessed on the specific problem instances. This design process must be repeated if results are not satisfactory or in case the problem configuration has changed. It is a very laborious and time expensive task, particularly considering the vast amount of methods and different design options. Thus, several approaches for automating the design of heuristic-based algorithms have been developed over the last years.

Methods for automating the configuration and generation of algorithms can be classified into: parameter configuration approaches (e.g., iRace [1]) and hyper-heuristics (HH) approaches, the latter are high-level methodologies for selecting or generating heuristics (be it simple heuristics or complete algorithms) [2]. In particular, HH for algorithm generation typically use Genetic Programming (GP) and favors a great spectrum of different combinations of components and thus different algorithm designs.

Several Genetic Programming Hyper-Heuristics (GPHH) have been applied to generate algorithms, such as tree-based GP [3], Grammatical Evolution (GE) [4], and Cartesian Genetic Programming (CGP) [5]. Although they produce competitive or even better algorithms than human-design ones, the parameter configuration is often neglected or considered to a limited extent (except the GE [4]). Since heuristic-based methods are sensitive to their settings, both the search for the most suitable algorithm design and the parameters of generated algorithms are crucial tasks to be performed by an automated algorithm configuration.

Thus, this work proposes a way of integrating parameter configuration into a GPHH, in particular, a CGPHH. Different from other GP approaches, CGP does not suffer from bloat, has an efficient decoding and fast convergence [6] and, has already been used to evolve algorithms for different problems. In a recent work [5], it was used to generate algorithms for the production lot-sizing problem (LSP), a crucial decision in production planning, that seeks to determine optimal production quantities that satisfy the demand, for each period, at minimal costs.

The investigation of automated configuration of heuristic-based algorithms for LSP is particularly important, as changes in the planning requirements or production configuration can affect the model, and consequently require a re-configuration of the methods used to solve them. With the introduction of reconfigurable and flexible production [7], the environment becomes even more susceptible to changes.

The multi-level capacitated lot-sizing problem (MLCLSP) is a particular problem within the general LSP models. Due to its multi-level structure and capacity restrictions, it portrays one of the most difficult problems in lot-sizing. Thus, this paper applies the proposed method to generate parametrized algorithms to solve the MLCLSP. The specific objectives are:

1) To integrate the parametrization of generated algorithms with the CGPHH.
2) To use the approach to evolve algorithms and their parametrization for the MLCLSP.
3) To analyze the contribution of parameter configuration on the evolution of algorithms.
4) To analyze the performance of the algorithms derived by the CGPHH with parameter configuration and compare them with the performance achieved without the parameter configuration.
Section 2 presents an overview of GPHHs, CGP, and introduces the approach for integrating parameter configuration with the CGPHH. Section 3 introduces the MLCLSP and gives an overview of related heuristic-based approaches. In section 4, the configuration of experiments is explained and results are analyzed. Finally, in section 5, conclusions are drawn and future research directions are pointed out.

II. CGP-BASED HYPER-HEURISTIC FOR ALGORITHM CONFIGURATION

Hyper-Heuristics are high-level methodologies that ultimately aim at increasing the generalization ability of heuristic-based methods. They are employed either to automatically select heuristics or algorithms from a set of existing ones or generate heuristics or algorithms from their components. Within generative HHs, Genetic Programming (GP) is the key ingredient and has been widely used because it can combine the components in a multitude of different forms, enabling the discovery of novel heuristic-based methods that can outperform hand-crafted methods for a given problem. GP-based Hyper-Heuristics (GPHHs) favor an ample design space characterization by the GP method employed, the structure of generated heuristic methods (e.g., simple heuristics, Genetic Algorithms), the set of components (i.e., the operators, variables, and constants that define the function and terminal sets), and how the performance of generated heuristic methods is measured (i.e., the fitness function) [8].

Among the GP methods, Cartesian Genetic Programming (CGP) does not require any parsimony pressure to control bloat, has fast convergence, and efficient decoding [6]. In contrast to the generally used tree representation, CGP uses direct acyclic feed-forward graphs to represent programs. The genotype represents the inputs and outputs for each node within a two-dimensional grid of function nodes, which can be connected to the preceding function or input nodes. Fig. 1 illustrates a CGP representation of an incumbent algorithm. In this example, all functions have arity one (i.e., has one argument/input), and the program has one output only. In order to use CGP, it is necessary to set: (i) the number of columns \( c \) and rows \( r \) of the grid; (ii) the maximum number of levels-back \( l \) any two connected nodes can be apart from each other; (iii) the function set, their respective functions arities and ids in the lookup table; (iv) the terminal set; (v) the definition of the fitness function for the generated programs (items iii-v are also required by others GP methods). Additionally, the genotypes are usually evolved by a (1+4) Evolutionary Strategy (ES) [9].

Even though the number of columns and rows defines fixed grid dimensions, the CGP can generate programs with variable number and combinations of operators (i.e., function nodes). The actual program (i.e., the phenotype) is decoded by traversing all (active) nodes that connect the program output(s) to the input node(s). In case \( l \) is equal to \( c \) (i.e., a node can be connected to any preceding function and input nodes), programs can vary from zero to the maximum number of nodes. Nodes that are not connected to the program output(s) are called introns (i.e., inactive nodes). The redundancy of nodes and the presence of introns in the genotype are of high importance for the performance of the evolutionary process [10].

A. CGP-based Hyper-Heuristic

CGP has been already used as a HH method in previous works to solve the Boolean Satisfiability Problem (SAT) [11], the Traveling Salesman Problem (TSP) [12, 13], and the Multi-level Capacitated Lot-sizing Problem (MLCLSP) [5]. Although CGP has been adapted to incorporate iterative flows [13], most of the existing approaches employ the standard CGP method, however using different algorithm templates and function and terminal sets.

In this work, the CGPHH employs the standard CGP to evolve algorithms and includes the selection and replacement operators in the function set. It allows multiple occurrences of selection operators within the generated algorithm, i.e., multiple sampling of individuals at different points of the evolutionary process. Additionally, it is possible to generate algorithms that correspond to an interleaving of two distinct evolutionary processes, i.e., contains one or more sequences of operators encompassed by selection and replacement operators. Additionally, the environment variable passed to the execution of each operator, is composed of the current population, the pool of parents (i.e., individuals that have been selected for breeding), and the pool of offspring (i.e., individuals that have been changed through mutation or crossover).

Although all operators receive the environment as argument for their execution, each one operates on a different list of individuals (TABLE I). A crossover takes the individuals from the pool of parents and generates the offspring in the pool of offspring. A mutation is applied to all individuals in the pool of parents and the pool of offspring. Mutated parents are included in the pool of offspring, while the offspring are updated with their mutated counterparts. Local search and mathematical heuristics try to improve the best individual from the current environment. Replacement updates the current population with the individuals from the pool of offspring, according to the replacement type, and removes all individuals from the pool of offspring. The fitness function is only computed when required, e.g., to identify the best individual.

The generation of algorithms follows the template depicted in Fig. 2. The current population (populated by an initialization...
B. CGPHH with Parameter Configuration (CGPHH-PC)

The proposed approach comprises of two tasks: (i) finding a good combination of algorithm components and (ii) searching for a proper parametrization of the generated algorithms. While the CGP evolves the algorithm’s composition, the parameter-configuration process assigns parameter values to the generated algorithms and optimizes the parameters of the best algorithms found so far. In addition, idle computational-resource(s) that have been allocated to the CGP are shared with the parameter-configuration process and used for evaluating further parametrizations. Computational resources become idle when a neutral drift occurs, i.e., the phenotype is not affected by changes in the genotype, resulting in an already evaluated algorithm. The presence of introns causes neutral drifts and it has been proved beneficial for the diversity of the generated programs [10]. In order to identify individuals with equal phenotypes and reuse computational resources, a cache mechanism is employed.

Fig. 3 depicts how the parameter configuration is integrated with the CGP. At each CGP generation, all the produced algorithms plus the best individual – in case of a (1+\lambda) ES, the parent – are passed to the parameter configuration processes, which consists of (i) defining the next parameter values to use for a given algorithm, (ii) evaluating the parametrized algorithm on the training instances, and (iii) updating the model used for generating the next parameter values based on the performance of the parametrized algorithm (necessary when using a model for the parameter configuration). The algorithm considered for parameter configuration depends on the current individual being evaluated. New parameterizations are sampled for the algorithms represented by the best or newly generated individuals. In case an individual has already been evaluated (i.e., it is in the cache), a new parameter setting is sampled for either the cached or the best individual. This strategy aims at giving a configuration budget to both cached and best individuals, hopefully improving either performance and driving them to achieve the best overall fitness.

The proposed approach enables the use of a variety of methods for searching better parameter settings for a given algorithm, such as sampling techniques (e.g., Latin hypercube, orthogonal, random), optimization methods (e.g., local search methods), learning and statistical models (e.g., reinforcement learning). Since the algorithms change over time, the employed method should take this aspect into account (either by adapting its model or by constructing a new one). This design flexibility allows adjusting the parameter configuration approach for the needs and complexity of the problem under consideration.

In order to assure that neutral drifts are not impacted by the integration of the parameter configuration approach, the CGP
individual and its update procedure have been adapted. Besides
the genotype and fitness, individuals also store the parameter
setting used. The best individual as a whole is replaced if an
offspring has equal or better fitness. Otherwise, if an offspring
has the same phenotype of the best individual, only the genotype
is replaced.

III. MULTI-LEVEL CAPACITATED LOT-SIZING PROBLEM

The MLCLSP is a generalization of lot-sizing problems and
one of the most difficult models of central production planning.
It takes several characteristics of the production environment
into account, thus reflecting the real-world closer and producing
more feasible plans. The production is modeled as a sequence of
production levels that transform raw materials into one or more
intermediate products, ultimately converting them into the final
products. Additionally, MLCLSP model considers dynamic
demand for final products, multiple final products, multiple
machines, capacity restrictions regarding the workers/machines,
costs and time to change the production setup between different
products, and inventory holding costs. Due to fluctuations in
demand and limited capacities, final products can be produced
and stocked to satisfy future demand. Besides that, some
variations of the MLCLSP consider backorders, overtime costs,
and setup carryover. The goal is the determination of the optimal
amount of products to produce at each period to satisfy the
demand and minimize costs. The standard MLCLSP is NP-
Complete [14].

Even though exact methods have also been applied to solve
MLCLSPs, most methods employed are heuristic-based, so that
feasible, near-optimal solutions can be found even for large
problem instances (e.g., genetic algorithms, tabu search,
relaxing or fixing variables of the mathematical programming
model, greedy heuristics) [15]. Among the solution approaches,
the hybridization of meta-heuristics and mathematical heuristics
has led to very good results. These hybrid methods, a.k.a.
matheuristics, take advantage of the good exploration
capabilities of meta-heuristics and the powerful exploitation
of mathematical heuristics to search for optimal solutions.

The fix-and-optimize (FaO) mathematical heuristic has been
often employed by matheuristics applied to MLCLSP [16–18].
FaO attempts to find optimal solutions by decomposing the
problem into sub-problems and iteratively solving them. The
sub-problems are defined by fixing some setup variables to
constant values (i.e., a binary matrix – products x periods – if
one, a product is produced in that period, incurring setup costs;
and zero, otherwise). Fig. 4 illustrates the FaO heuristic. A
window determines which values are set as variables and
constant. Then, the sub-problem is optimized and variables fixed
with the optimal values found. Subsequently, the window is
rolled (over the products or periods) and the process repeats until
a stop criterion is satisfied. Several variants of the FaO heuristics
have been proposed and decompose the problem based on the
products, periods, production processes, resources, or a
combination of them [16–19].

The multi-population genetic algorithm with fix-and-optimize
(MGAFO) [16] is a prominent matheuristic method for
MLCLSP. It has produced very good solutions for the
considered problem instances and contains several components,
besides the FaO, that fits well in the CGPHH framework. Since
most components used in this work were extracted from
MGAFO, this approach is further explained and latter compared
against the generated algorithms. The MGAFO uses three GA
populations, hierarchically organized in a quaternary-tree
structure of depth three: the root is the best individual of
the population and the leaves present individuals with worse fitness
than the respective parent nodes. It uses the setup variables as
solution representation and initializes them with a hot-start
procedure. The fitness is computed by solving the linear
programming model with CPLEX, given the setup variables.

Parents are selected based on their tree-hierarchy: only the
root and its immediate children are eligible for mating. An
offspring only enters the population if its fitness is better than
that of its parents. The evolutionary process is performed by four
crossover and three mutation variants, which are randomly
selected at each iteration. The crossover variants are (i) uniform;
(ii) sector: the setup matrix is randomly divided into four
partitions and interchanged between the parents at random; (iii)
one point period; and, (iv) one point product. Mutation variants
comprise (i) random one flip; (ii) random two flips for the same
period; (iii) random two flips for the same product; and (iv)
random two flips.

After populations have converged, the best individual of
each population migrates to the next population in a round-ring
fashion, the FaO used with periods and products attempts to
improve the best individual among all populations. Next, the
improved individual is inserted into the populations. If the FaO
failed at improving the best individual, its window size is
increased (i.e., a larger set of variables is considered). After this
process, the evolutionary process starts again and this cycle
continues until the execution time is reached.

IV. EXPERIMENTAL RESULTS

In order to enable comparison, the experimental setting is the
same as defined for the CGPHH with static parameter values
(CGPHH-S) [5]. The CGP is configured with: one row, 200
columns, levels back = 200, 1000 generations, a (1+4) ES with
4% mutation rate to evolve algorithms, which are in turn
executed five times for each training instance and their fitness
computed as the mean fitness over the different instances and
runs. The same mathematical models for the fitness function
and FaO heuristics are used, as well as the same version of CPLEX,
running with two threads limit. Thus, at maximum, a total of
5000 different combinations of algorithms and parameters are
tested at each CGP execution, due to the parameter-
configuration procedure. The time budget of the algorithms
is 180s. The CGPHH-PC is also implemented in Java 8 and uses
The use of any other more advanced method for parameter bound on the performance evaluation of the CGPHH-PC. The most naïve method to use, it can represent a theoretical lower-bound for the generated algorithms. Since this is the configuration space of those heuristics, better heuristic designs are achieved when the heuristics, e.g., to configure different behaviors of rolling windows product- and period-wise. By increasing the number of columns (i.e., periods) or rows (i.e., products), fix all but the selected periods and products as constants, and tries to optimize the remaining set of setup variables. This process repeats until an improved solution is found or the maximum number of trials is reached. If no enhanced solution is found, the number of trials increases. Three non-deterministic FaO approaches have been conceived: product-wise, period-wise, and product- and period-wise.

The parameters of each operator used in the function set have been extracted and modeled, as shown in TABLE II. The permissible values of numeric and integer parameters are modeled as \((\text{lowerBound}, \text{upperBound}; \text{interval}) [\text{defaultValue}]\), where values between the bounds (inclusive) are discretized in equal intervals. Several parameters are used by the mathematical heuristics, e.g., to configure different behaviors of rolling windows product- and period-wise. By increasing the configuration space of those heuristics, better heuristic designs might be discovered.

A random sampling without repetition is used to set the next parameter setting for the generated algorithms. Since this is the most naïve method to use, it can represent a theoretical lower-bound on the performance evaluation of the CGPHH-PC. The use of any other more advanced method for parameter configuration will likely perform the search in the parameter space more effectively. New generated algorithms are randomly assigned with the default parameter-values or the parameter setting of the best individual so far. Further configurations for the best or cached algorithms are assigned so that the combination \((\text{algorithm}, \text{parameters})\) is unique.

![Fig. 5. Different parametrizations tested during the execution of CGPHH-PC](image)

TABLE II. LIST OF PARAMETERS TO BE CONFIGURED.

<table>
<thead>
<tr>
<th>Mathematical Heuristics</th>
<th>FaO non-deterministic</th>
</tr>
</thead>
<tbody>
<tr>
<td>prod. min. window (2, 4; 1) [2]</td>
<td>min. products (2, 4; 1) [2]</td>
</tr>
<tr>
<td>prod. max. window (4, 7; 1) [7]</td>
<td>max. products (4, 7; 1) [7]</td>
</tr>
<tr>
<td>prod. inc. window (1, 3; 1) [1]</td>
<td>inc. products (1, 3; 1) [1]</td>
</tr>
<tr>
<td>perio. min. window (2, 5; 1) [2]</td>
<td>min. periods (2, 5; 1) [2]</td>
</tr>
<tr>
<td>perio. max. window (5, 10; 1) [7]</td>
<td>max. periods (5, 10; 1) [7]</td>
</tr>
<tr>
<td>perio. inc. window (1, 3; 1) [1]</td>
<td>inc. periods (1, 3; 1) [1]</td>
</tr>
<tr>
<td>rolling span (1, 3; 1) [1]</td>
<td>max. no. trials (2, 8; 1) [4]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FaO Convergence</th>
<th>improvement rate (0.005, 0.05; 0.005) [0.05]</th>
</tr>
</thead>
<tbody>
<tr>
<td>n. evaluations</td>
<td>(1000, 5000; 1000) [1000]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>n. of indiv. (20, 80, 30) [50]</td>
<td>prob. mutation (0.005, 0.1; 0.01) [0.015]</td>
</tr>
<tr>
<td>init. meth. {MGAFO, random=MGAFO} [MGAFO]</td>
<td>mutation rate (0.75, 1; 0.05) [0.95]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Selection</th>
<th>Crossover</th>
</tr>
</thead>
<tbody>
<tr>
<td>tourn. size (5, 20; 5) [10]</td>
<td>unif. crossover (0.5, 0.9; 0.05) [0.6]</td>
</tr>
<tr>
<td>n. of indiv. (2, 4; 1) [2]</td>
<td>crossover rate (0.75, 1; 0.05) [0.95]</td>
</tr>
</tbody>
</table>

TABLE III. SUMMARY OF CGPHH-PC RESULTS.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ALLLS</td>
<td>499</td>
<td>4501</td>
<td>141942.5</td>
<td>-1.58</td>
<td>0.0,0,1,0,0</td>
</tr>
<tr>
<td>ALLNR</td>
<td>1538</td>
<td>3462</td>
<td>141188.8</td>
<td>-2.11</td>
<td>0.0,0,3,0,0</td>
</tr>
<tr>
<td>CROSLS</td>
<td>1752</td>
<td>3248</td>
<td>141465.3</td>
<td>-1.92</td>
<td>1,1,0,3,2,1</td>
</tr>
<tr>
<td>CROSNR</td>
<td>2243</td>
<td>2757</td>
<td>141182.5</td>
<td>-2.11</td>
<td>2,4,0,7,0,1</td>
</tr>
<tr>
<td>LS</td>
<td>1195</td>
<td>3805</td>
<td>141002.9</td>
<td>-2.24</td>
<td>0.0,0,6,3,0</td>
</tr>
<tr>
<td>LNSR</td>
<td>877</td>
<td>4123</td>
<td>140961.9</td>
<td>-2.26</td>
<td>0.0,0,11,0,0</td>
</tr>
<tr>
<td>MUTLS</td>
<td>1521</td>
<td>3479</td>
<td>141341.5</td>
<td>-2.00</td>
<td>1,0,1,4,2,1</td>
</tr>
<tr>
<td>MUTNR</td>
<td>1582</td>
<td>3418</td>
<td>140929.9</td>
<td>-2.29</td>
<td>1,0,1,5,0,1</td>
</tr>
</tbody>
</table>

A sequence regards to: Selection, Crossover, Mutation, FaO, non-deterministic FaO, Replacement.
algorithms resulting from the CGPHH-PC have better fitness for the training instances compared to the outcome of the CGPHH-S, when parameters were set to default values.

The number of distinct algorithms (i.e., distinct sequence of operators) generated per experiment varies considerably. This is likely attributed to the particular graph-structures and neutral drifts. The generation of fewer distinct algorithms, on the other side, implies giving a higher budget for parameter configuration, which in turn can significantly improve the algorithm’s performance. For instance, LSNR examines only 877 different algorithms, the second-lowest amount, and was able to find a good parametrized algorithm that produced one of the best fitness for the training instances.

TABLE IV. shows the gap and respective fitness standard-deviation of the MGAFO (re-executed on the same computational environment) and the best algorithms generated by the CGPHH-PC for the instances of the test set. MUTNR, LSNR, MUTLS, and CROSNR present a higher overall improvement and less fitness deviation compared to MGAFO, while ALLLS failed to improve the overall result, but achieved the best improvement for three instances. Only MUTNR and MGAFO consistently improve solutions for all cases. The Friedman’s aligned rank test for multiple comparisons rejects the hypothesis that all algorithms are equal (p-value=0.0072), but the Nemenyi posthoc test (a non-parametric equivalent of the Tukey test) does not show a significant difference between the algorithms at 5% significance level (alpha = 5%).

**Fig. 6.** Fitness variation of MGAFO and best algorithms generated by CGPHH-PC and CGPHH-S for each problem instance.

<table>
<thead>
<tr>
<th><strong>MGAFO</strong></th>
<th><strong>ALLLS</strong></th>
<th><strong>ALLNR</strong></th>
<th><strong>CROSL</strong></th>
<th><strong>CROSNR</strong></th>
<th><strong>LS</strong></th>
<th><strong>LSNR</strong></th>
<th><strong>MUTLS</strong></th>
<th><strong>MUTNR</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>G511132</td>
<td>-0.01 (637)</td>
<td>-0.65 (0)</td>
<td>-0.37 (0)</td>
<td>-0.47 (575)</td>
<td>-0.70 (0)</td>
<td>6.37 (30513)</td>
<td>-0.54 (0)</td>
<td>-0.58 (667)</td>
</tr>
<tr>
<td>G511142</td>
<td>-0.11 (2265)</td>
<td>-1.25 (0)</td>
<td>-0.36 (988)</td>
<td>-0.70 (1201)</td>
<td>-0.31 (0)</td>
<td>-1.49 (1567)</td>
<td>-0.50 (0)</td>
<td>-1.17 (1958)</td>
</tr>
<tr>
<td>G512131</td>
<td>-0.52 (2666)</td>
<td>-1.43 (0)</td>
<td>-1.13 (0)</td>
<td>-0.86 (1099)</td>
<td>0.57 (0)</td>
<td>-0.72 (1146)</td>
<td>-1.13 (0)</td>
<td>-0.85 (456)</td>
</tr>
<tr>
<td>G512142</td>
<td>-0.60 (1190)</td>
<td>-1.73 (0)</td>
<td>-1.55 (0)</td>
<td>-1.81 (2132)</td>
<td>1.94 (1801)</td>
<td>-1.08 (1377)</td>
<td>-1.52 (0)</td>
<td>-1.09 (1338)</td>
</tr>
<tr>
<td>G521131</td>
<td>-0.67 (962)</td>
<td>-1.69 (0)</td>
<td>-0.81 (0)</td>
<td>-0.67 (1030)</td>
<td>0.15 (292)</td>
<td>-1.04 (1131)</td>
<td>-1.05 (0)</td>
<td>-0.92 (1342)</td>
</tr>
<tr>
<td>G521142</td>
<td>-0.01 (641)</td>
<td>-0.06 (0)</td>
<td>-0.32 (0)</td>
<td>-0.24 (747)</td>
<td>-0.32 (0)</td>
<td>-0.36 (720)</td>
<td>-0.58 (0)</td>
<td>-0.27 (544)</td>
</tr>
</tbody>
</table>

**Overall:** -1.05 | 0.87 | -0.62 | -0.85 | -1.23 | -0.79 | -1.53 | -1.29 | -1.57

* Instances used by the CGPHH-PC for evolving the algorithms.
The performance of algorithms varies significantly across different instances. The algorithms produced by CROSSLS, LS, and MUTLS include the non-deterministic FaO and present high deviation of fitness values. Instance-wise, at least one of the generated algorithms presented less variance and better mean fitness than the MGAFO, except for the instances K511132 and K522131. Algorithms generated by CGPHH-PC usually perform similar or better than those produced by CGPHH-S, particularly for instances with general production structure (starting with G). However, for K521132 and K521142 they are worse. Friedman’s aligned rank test applied to all algorithms and instances rejects the hypothesis of performance similarity (p-value=0.014). Nonetheless, the Nemenyi test does not show a significant difference between them.

A detailed analysis of the algorithms’ performance for different cutoff times is carried out for the eight best-ranked algorithms. TABLE V. presents the gap and rank of the different cutoff times. The performance over different time limits, ALLNR is the worst-performing algorithm (in the group of best algorithms only at 180s), followed by MGAFO (within the best algorithms at 10s and 180s). For short execution times, particularly for 20s and 30s, C+MH_01 and ALL_02 achieve very good results and a greater rank distance to other best-performing algorithms. Additionally, only MUTNR and LSNR make into the group of best-performing algorithms for all cutoff times, thus presenting the best anytime behavior among all algorithms.

V. CONCLUSIONS AND OUTLOOK

In this work, the CGPHH-PC is proposed: a CGPHH for algorithm generation with parameter configuration. It is based on the CGPHH with static parameters for generated algorithms (CGPHH-S) developed previously [5]. The approach to integrate parameter configuration into CGPHH framework is flexible and enables several methods for searching a better parameter setting for a given algorithm to be plugged-in, thus allowing adaptations of its design to the needs and complexity of the problem under consideration. The CGPHH-PC was used to design parameterized algorithms for MLCLSP, a complex problem with practical relevance in production planning, and employed a very naïve approach to configuring parameters, based on random sampling without repetition. Experiments were performed to analyze the contribution of the parameter algorithms. Although, the analysis for the 180 seconds run does not identify significant distinction, the rank difference between MGAFO and MUTNR (2.71) is very close to the critical difference (CD) computed by the method (2.79). Considering the performance over different time limits, ALLNR is the worst-performing algorithm (in the group of best algorithms only at 180s), followed by MGAFO (within the best algorithms at 10s and 180s). For short execution times, particularly for 20s and 30s, C+MH_01 and ALL_02 achieve very good results and a greater rank distance to other best-performing algorithms. Additionally, only MUTNR and LSNR make into the group of best-performing algorithms for all cutoff times, thus presenting the best anytime behavior among all algorithms.

TABLE V. RESULTS OF BEST ALGORITHMS FOR DIFFERENT CUTOFF TIMES.

<table>
<thead>
<tr>
<th>Time</th>
<th>MGAFO</th>
<th>MUTNR</th>
<th>LSNR</th>
<th>MUTLS</th>
<th>LS</th>
<th>CROSNR</th>
<th>C+MH_01</th>
<th>ALL_02</th>
<th>ALLNR</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>10s</td>
<td>-1.05 (6.10)</td>
<td>-1.57 (3.39)</td>
<td>-1.53 (3.53)</td>
<td>-1.29 (4.21)</td>
<td>-0.79 (4.89)</td>
<td>-1.23 (5.18)</td>
<td>-0.82 (5.92)</td>
<td>-0.86 (5.97)</td>
<td>-0.62 (5.79)</td>
<td>&lt;2.2E-16</td>
</tr>
<tr>
<td>120s</td>
<td>0.04 (6.74)</td>
<td>-1.14 (2.87)</td>
<td>-1.16 (3.95)</td>
<td>-0.96 (4.16)</td>
<td>-0.30 (5.16)</td>
<td>-0.82 (5.18)</td>
<td>-0.58 (5.37)</td>
<td>-0.56 (5.53)</td>
<td>1.02 (6.05)</td>
<td>0.01219</td>
</tr>
<tr>
<td>60s</td>
<td>6.92 (7.89)</td>
<td>1.34 (3.97)</td>
<td><strong>0.19 (3.21)</strong></td>
<td>5.03 (5.21)</td>
<td>20.58 (7.26)</td>
<td>6.36 (4.21)</td>
<td>0.08 (3.34)</td>
<td>-0.04 (3.82)</td>
<td>2.97 (6.08)</td>
<td>8.47E-10</td>
</tr>
<tr>
<td>30s</td>
<td>13.2 (6.52)</td>
<td>2.80 (4.10)</td>
<td>10.16 (3.97)</td>
<td>45.94 (6.42)</td>
<td>60.42 (8.42)</td>
<td>21.86 (4.84)</td>
<td><strong>0.64 (2.37)</strong></td>
<td>0.62 (2.63)</td>
<td>19.18 (5.71)</td>
<td>3.45E-13</td>
</tr>
<tr>
<td>20s</td>
<td>19.46 (5.58)</td>
<td>4.64 (3.66)</td>
<td>22.43 (4.05)</td>
<td>57.06 (6.84)</td>
<td>67.39 (7.53)</td>
<td>63.91 (7.16)</td>
<td>2.11 (2.31)</td>
<td><strong>1.71 (1.95)</strong></td>
<td>35.63 (5.92)</td>
<td>&lt;2.2E-16</td>
</tr>
<tr>
<td>10s</td>
<td>33.16 (2.68)</td>
<td>32.75 (3.68)</td>
<td>43.10 (4.18)</td>
<td>65.32 (4.63)</td>
<td>71.45 (5.79)</td>
<td>82.94 (7.87)</td>
<td>66.16 (4.71)</td>
<td>58.60 (5.76)</td>
<td>57.34 (5.68)</td>
<td>1.74E-08</td>
</tr>
</tbody>
</table>

Fig. 7. Parallel plot of overall gap obtained for the different cutoff times.

Fig. 8. Groups of algorithms based on the critical distance computed by the Nemenyi test. CD (critical difference); bar connects groups of similar algorithms (i.e., within the critical difference). 180s (top left), 120s (top right), 60s (middle left), 30s (middle right), 20s (bottom left), 10s (bottom right).
configuration and to contrast the performance of algorithms generated by CGPHH-PC with those produced by CGPHH-S.

The analysis of results indicates that the parameter configuration was able to find very good parametrized algorithms compared to the attribution of static default-values, even though the method employed for parameterization is quite rudimentary. Thus, it indicates that even better results can be achieved if more robust methods of parameterization are employed. The large number of different parameter configurations performed during the CGPHH-PC execution shows that it considerably succeeded in reusing computational resources, which would have been wasted due to neutral drifts. In addition, the non-deterministic FaO heuristics produced good results when executed together with deterministic FaO operators. However, algorithms that use these new operators have a higher standard deviation, given their non-deterministic nature.

All the best algorithms generated by CGPHH-PC were executed for a wide range of different parameter values. Although not statistically different from the algorithms generated by CGPHH-S, they have achieved a considerable overall-performance improvement. Additionally, the best algorithms of CGPHH-PC achieved the biggest improvement in comparison to the human-designed MGAFO for eighteen out of nineteen instances used in the test set. CGPHH-PC was also able to generate two algorithms (MUTNR and LSNR) with significant best overall-performance for any budget imputed on their execution time. Thus, the proposed approach is able to evolve algorithms that have a considerably higher fitness improvement and significant best overall-performance for different cutoff times.

Results also revealed that generated algorithms perform considerably different across the problem instances. Thus, an exciting research direction is an investigation of how features of MLCLSP impact the performance of algorithms. At first, an analysis of relevant features of MLCLSP should be carried out. Then, the acquired knowledge can be used to cluster the instances according to the features that have greater impact on performance and use the CGPHH-PC to generate parameterized algorithms automatically for the different groups of instances. The set of generated algorithms and problem features could then be used by an automatic, instance-based, algorithm-selection method. Furthermore, future research should focus on the comparison of different parameter configuration methods and evaluate their impact on the quality of generated algorithms and the overhead caused on the CGPHH-PC for different problem classes.

REFERENCES


