Hindawi
Scientific Programming
Volume 2018, Article ID 6971827, 13 pages
https://doi.org/10.1155/2018/6971827

Research Article

Novel Methods Generated by Genetic Programming for the Guillotine-Cutting Problem

Vittorio Bertolini,[1] Carlos Rey,[1] Mauricio Sepulveda,[2] and Victor Parada [1]

[1]*Informatics Engineering Department, University of Santiago of Chile, Santiago, Chile*
[2]*Informatics Engineering Department, San Sebastián University, Santiago, Chile*

Correspondence should be addressed to Victor Parada; victor.parada@usach.cl

Received 25 January 2018; Revised 2 August 2018; Accepted 7 August 2018; Published 2 September 2018

Academic Editor: Emiliano Tramontana

New constructive algorithms for the two-dimensional guillotine-cutting problem are presented. The algorithms were produced from elemental algorithmic components using evolutionary computation. A subset of the components was selected from a previously existing constructive algorithm. The algorithms' evolution and testing process used a set of 46 instances from the literature. The structure of three new algorithms is described, and the results are compared with those of an existing constructive algorithm for the problem. Several of the new algorithms are competitive with respect to a state-of-the-art constructive algorithm. A subset of novel instructions, which are responsible for the majority of the new algorithms' good performances, has also been found.

1. Introduction

Various industrial processes exist in which the raw material must be cut into smaller sections that must be assembled to produce the final product, as in the case of cutting plastics, glass, paper, and metals [1–3]. A typical case occurs in the wooden board cutting industry that requires efficient techniques to minimize the loss of material in furniture manufacturing. A piece of furniture is manufactured from rectangular pieces of wood cut from rectangular wooden plates by a saw which allows an end-to-end cutting of the plate [4, 5]. In turn, Park et al. [6] describe the situation that occurs during the manufacturing and cutting of glass. In such a case, a continuously produced sheet of glass is cut into large sheets, which in turn are cut into smaller rectangular pieces according to the customer requirements. The cut is made according to an optimal cutting pattern that minimizes wasted glass. These kinds of processes generated a family of stock cutting problems, which aim at determining the best method of using raw materials [7].

Often approached from a combinatorial optimization perspective, cutting problems represent an intellectual challenge because of the computational difficulty that arises when attempting to solve them [8]. A particular case is the constrained two-dimensional guillotine-cutting problem studied in this paper, which focuses on cutting rectangular plates [9, 10].

The statement of the problem considers a rectangular plate of length L and width W that must be cut into a set of m small rectangular pieces $p_1, p_2, \ldots, p_m$ of sizes w_i and l_i and area $s_i = w_i \cdot l_i$ such that $w_i \leq W$ and $l_i \leq L$ for every $i \in P = \{1,2,\ldots,m\}$. A limit $b_i > 0 \, \forall \, i \in P$ that corresponds to the number of times that piece i, with profit $v_i > 0$, can be cut from the rectangular plate is considered. A cutting pattern is a feasible configuration of pieces to be cut from the plate. The geometric feasibility of the cutting pattern considers that (i) all cuts must be of the guillotine type, (ii) there should be no overlap among the pieces that constitute the pattern, and (iii) the pieces must be positioned in a fixed orientation. Defining $x_i \in Z_0^+$ as the number of times that a piece of type i is found in a pattern, the problem lies in determining a cutting pattern with a maximum value $z = \sum_i v_i x_i$ such that $0 \leq x_i \leq b_i, \ \forall \ i \in P$. Following the commonly used notation [11, 12], the problem corresponds to the constrained weighted version (CW_TDC). Conversely, following the classification of Wäscher et al. [7], this case corresponds to

a two-dimensional, rectangular, single large object placement problem.

The problem has been studied not only for its impact on the optimization of raw material use in industrial processes but also for the computational difficulty that arises when attempting to solve it by exact methods. Three initial approaches that are based on the dynamic programming formulation (DPF) have generated these impacts [13]: the solution space formulation using graphs (GF), which produces search methods based on trees [14, 15], and the constructive approach (CF), which allows combining rectangles in an increasing manner while retaining the feasibility of the guillotine cuts [16]. These fundamental methods have also been improved by incorporating changes that allow optimal solution searching to be performed with a better computational performance [17–21]. The most efficient results have been achieved with a hybridization of the fundamental ideas, and thus, search methods based on trees have been designed using the hybridization of GF with DPF, which have allowed solving for small- and medium-sized instances of the problem [10, 22–25].

For large instances of the problem, the heuristic combinations are more complex. Álvarez-Valdés et al. [9] combine constructive procedures that allow for the determination of the upper bounds with path relinking, and they use the GRASP and tabu search [26] metaheuristics as base algorithms. The authors also present a constructive algorithm (CONS) whereby at each iteration, a new piece is assigned over a rectangle that is dynamically updated. Conversely, Morabito et al. [27] generated a hybrid search strategy by combining depth-first searching with hill climbing. A third hybridization level is incorporated into the method by combining it with an algorithm that solves the problem using a sequence of one-dimensional knapsack subproblems [28]. Hybridization between simulated annealing and evolutionary algorithms, considering both GF and CF approaches, has also been explored numerically to solve different sized problems [29].

During the last four decades of research on the CW_TDC problem, researchers have performed computational studies of some specific hybridizations of the existing methods to obtain better performance in terms of both the computational time and the quality of the obtained solution. However, other potential hybridizations have not been explored. This paper proposes that the exploration of those methods can be accelerated with the support of evolutionary computation, specifically using the same ideas that support genetic programming [30, 31]. This discipline allows the step-by-step combination of complex structures by following the principles of Darwinian evolution. By combining the elemental components of a constructive algorithm to solve the CW_TDC problem with some components specifically generated for this problem, novel and new algorithms can be produced and analyzed to discover sets of instructions having some logic that may give rise to new algorithmic ideas. In this paper, we describe the performance of a set of created algorithms that are compared with two known heuristics for the problem, estimating their similarities and relative computational designs. A subset of the elemental algorithmic components was selected from the constructive algorithm CONS, aiming to produce other similar algorithms following the same idea.

2. Design of Elemental Algorithmic Components

The algorithms are generated by genetic programming (GP), a particular technique in the field of evolutionary computation. The latter is an area of knowledge that refers to the study of methods inspired by Darwinian evolution to solve problems in science and engineering [31, 32]. In GP, the elementary components of computer programs are gradually assembled to generate a piece of computer code that is responsible for performing a specific task. The pieces of code are represented by syntactic trees. This way of automatically producing a computer program starts with some high-level specifications and gradually generates the code for the intended task. Populations evolve gradually through the application of selection, variation, and reproduction operators, so that, population after population, the structures become increasingly specialized in their specific task. The elementary components can be functions such as the typical instructions *while*, *if-then*, *or*, *and*, among others. Also, GP can consider specific functions for the intended task. When algorithms are automatically generated for a combinatorial optimization problem, such elementary components can be ad-hoc heuristics for the problem. Figure 1 illustrates the generation of algorithms for the CW_TDC.

Algorithms for CW_TDC are gradually built by an evolutionary process. To perform this task, each algorithm is represented as a tree of instructions, where the intermediate nodes are high-level instructions, and leaf nodes corresponding to problem-specific functions are entrusted to build the layout. The process is outlined in Figure 1. From a population $P(k)$ containing a fixed number of trees, the implementation of the selection, crossover, and mutation operations generates a new population $P(k+1)$. To evaluate the performance of each tree, the fitness evaluation module uses a set of adaptation problem instances available in the literature for the CW_TDC problem. The instructions are executed sequentially, traversing the tree with an in-order, depth-first search. The best tree found during the evolutionary process is stored and subsequently decoded into the corresponding pseudocode.

2.1. Basic Definitions. Several definitions are needed to implement this approach. The first is the definition of the sets of high-level instructions and of problem-specific functions for the CW_TDC problem. It is also necessary to have sets of adaptation and testing problem instances. Finally, a fitness function, which is responsible for guiding the evolutionary process, is required. The basic idea is to generate a new algorithm from a well-known, previously existing heuristic for a problem and from there, start an improvement process. In this sense, we consider CONS [9], which efficiently solves several problem instances of CW_TDC as the reference algorithm. First, such an algorithm is decomposed into its elemental components, and

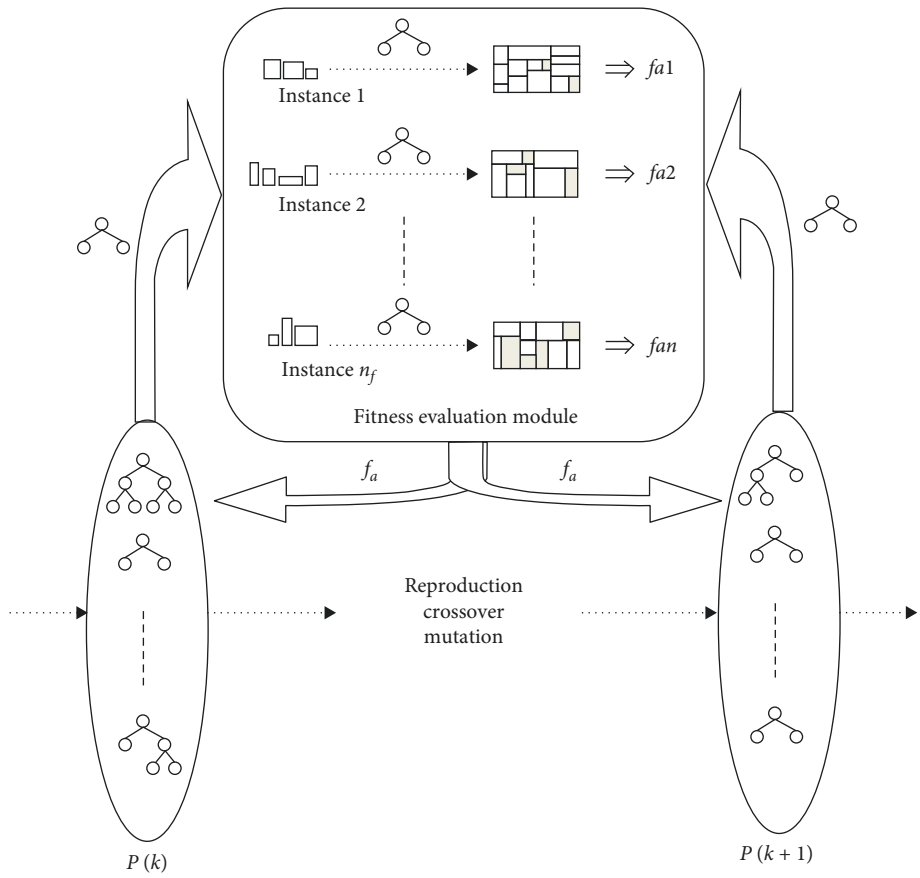

Figure 1: Evolution of algorithms.

then, other similar algorithms are created. The new algorithms are the result of the combination of those elemental components. The CONS constructive algorithm has a main cycle through which, at each iteration, a new piece is assigned over a rectangle R_k with dimensions W_k and L_k. A guillotine-type cut generates four new, smaller external rectangles R_k^1 and R_k^2 or R_k^3 and R_k^4, depending on whether the cut is vertical or horizontal. Those rectangles are stored to be cut in the following iteration by the same logic (Figure 2). The process stops when it is no longer possible to assign a piece to the stored rectangles, and these pieces are considered as a loss of material.

The piece chosen for assigning in a rectangle R_k is the one that generates the maximum estimated profit. This value is calculated as the profit of the piece plus the sum of the profits when assigning, in decreasing order of $r_i = v_i/s_i$, the available pieces in the rectangles R_k^1 and R_k^2 or R_k^3 and R_k^4. Specifically, this profit is estimated using an algorithm BK_1 to solve the knapsack problem [33]. In this case, the elements of the knapsack are the pieces; the weight corresponds to the area, and the capacity corresponds to the area of each rectangle R_k^1 and R_k^2 or R_k^3 and R_k^4. A procedure BK_2 is also defined, which, unlike BK_1, inserts the first piece available as many times as it fits in the rectangle and applies the estimates based on the knapsack problem for spaces not occupied by the first piece.

In addition to BK_1 and BK_2, we define two new procedures, namely, BK_3 and BK_4, to calculate the estimated profit in the outer rectangles. BK_3 is a variant of BK_2 that assigns available pieces to the horizontal base of the rectangle, in decreasing order of $r_i = v_i/s_i$, until there is no more space for a piece to fit (Figure 3). The profit is then estimated using BK_1 in the rectangles R^1, R^2, ..., R^q, which are generated by considering a horizontal line drawn from the widest piece already assigned.

BK_4 is defined by locating the pieces sequentially to different rows of the rectangle, as defined in the work of Coffman et al. [34], for a strip of infinite length. In this case, the height of each row is given by the tallest piece when assignment takes place from the bottom left corner without going beyond the limit L_k (Figure 4(a)). The pieces are assigned following the decreasing order of $r_i = v_i/s_i$. During the piece-assigning process, the pieces that do not fit in the rectangle being formed (between the height of the row and width W_k and between the sum of the lengths of the pieces assigned to the row and L_k) are ignored (Figure 4(b)). A new row is created when the next piece does not fit along the length but does fit in an upper level.

The algorithms to be constructed must operate on three data structures. First, a list of pieces available (LPA) stores the pieces remaining at each stage. Second, a list of rectangles (LR) is completed as the algorithm advances, with rectangles R_k still to be processed. At each step, there is an active rectangle being processed. At the beginning, the list contains only the original rectangle. Those rectangles in which no

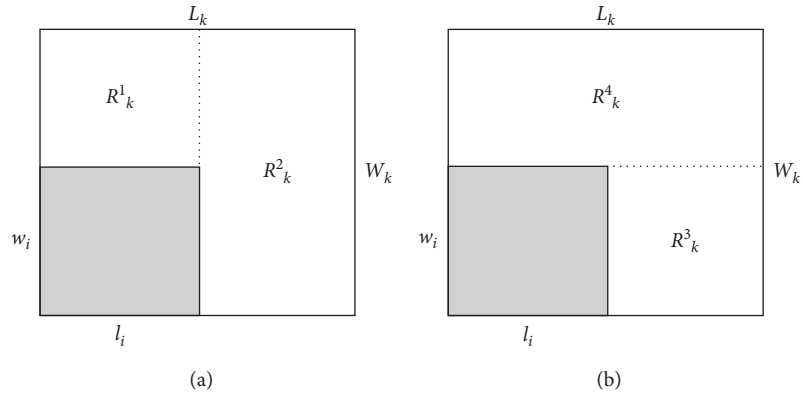

Figure 2: (a) Vertical and (b) horizontal cuts.

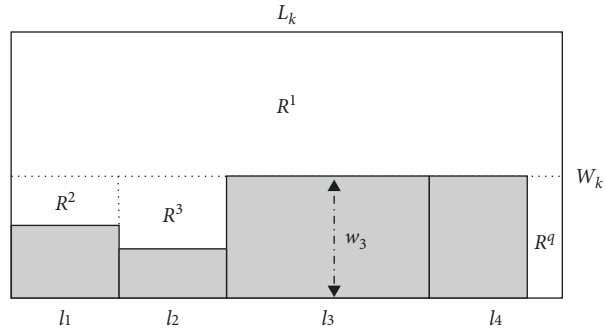

Figure 3: Rectangles for evaluating BK_3: R^1, R^2, R^3, and R^q.

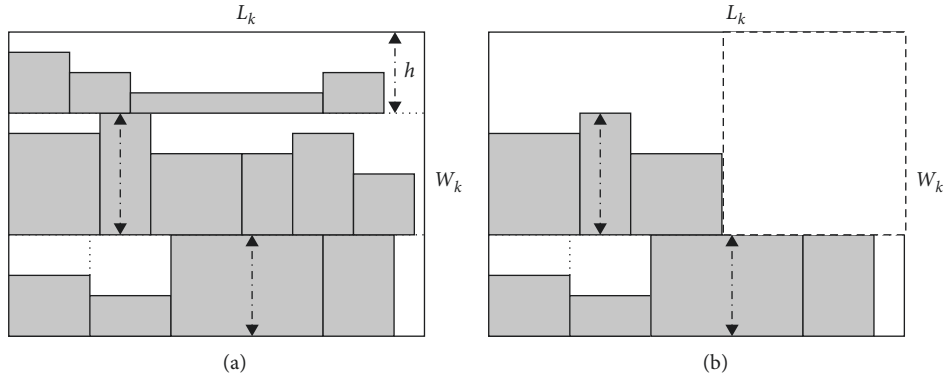

Figure 4: Geometry to obtain the upper bound BK_4 in rectangle R_k. (a) Completed and (b) in process.

available piece fits are considered losses. Finally, a stack of blocks (SB) contains blocks of pieces constructed using vertical or horizontal joining operations either between pieces or blocks according to the Wang heuristic [16]. Supported by this stack, it is not necessary to assign the selected piece immediately, as occurs with CONS, but it can be stored and therefore joined with the other pieces, forming blocks whose size limit is given by the processed rectangle R_k.

2.2. Definition of Functions. The elemental components of the algorithms to be produced are translated into two sets of functions. The first set contains the basic instructions in most computer languages and is defined with the parameters P_1 and P_2 as integer variables, considering that a value greater than 0 corresponds to the "true" logical value and that a value equal to 0 corresponds to "false." All functions return true or false, and they are While (P_1, P_2), IfThen (P_1, P_2), Not (P_1), And (P_1, P_2), Equal (P_1, P_2), and Or (P_1, P_2).

The second set contains specific functions for the CW_TDC problem that are sufficient to allow for the reconstruction of the reference algorithm. Figure 5 depicts the function's mode of operation. It shows a rectangle R_k receiving a block and therefore giving rise to two new

rectangles to be placed in LR. Functions selecting a piece from LPA to be combined with the already existing blocks in SB are also sketched. The following variables are necessary:

(i) *OP*: this variable stores the order of the pieces used in the BK_i procedures. The default value is *Val-Standard_v/s*, indicating that a decreasing order $r_i = v_i/s_i$ must be used.

(ii) *BKj*: this variable stores the BK_i procedure to be used. The default value is *ValStandard_BK*$_1$, indicating that BK_1 must be used.

(iii) R_k: this variable stores the mechanism for selecting the next rectangle R_k to be assigned to LR. The default value is *ValueStandard_BK*, indicating that the estimator indicated in *BKj* must be used.

(iv) *SU*: this variable indicates that the horizontal or vertical joining process for the next piece to be placed in SB must be stopped.

Then, the following specific functions are defined:

(i) *Add-p*(): a function that inserts the available piece into SB that maximizes the estimated profit using BK_1 over the active rectangle R_k. The function returns the piece number.

(ii) *Cut*(): this function assigns a block from SB to the active rectangle R_k and deletes it from LR. Two rectangles are generated, either R_k^1 and R_k^2 in the horizontal case or R_k^3 and R_k^4 in the vertical case (Figure 2). The pair of rectangles that generates the largest estimated profit using BK_1 is selected. The selected rectangles are stored in LR. Then, a new rectangle R_k in LR is activated, which corresponds to the rectangle with the largest estimated profit, using BK_1 with the available pieces. The function always returns 1 (true).

(iii) BK_2(): a function that acts as a flag to indicate that, in the next execution of the *Cut* or *Add-p* functions (whichever occurs first), the indicator BK_2 must be used. It always returns 2 (true).

(iv) *MINWL_WASTE*(): this function acts as a flag to indicate that, in the next Cut() execution, the rectangle R_k with the smallest area must be selected. It returns 1 (false) or 2 (true) if there is a rectangle R_k.

Based on the set of functions and variables *OP*, *BKj*, *Rk*, and *SU*, the CONS algorithm can be reconstructed as in Algorithm 1.

To provide greater variability for algorithm construction, new specific functions are generated based on the following four strategies:

(a) Provide greater freedom to join a piece with a block in the SB, according to the horizontal or vertical construction method. With this strategy, the pieces can be assigned as they are selected, and they can also be joined, forming a block to be assigned later.

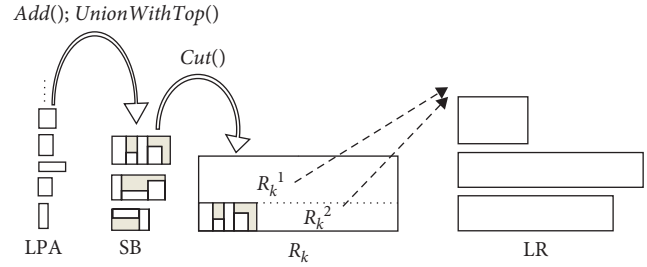

Figure 5: Function operations on the lists LPA, SB, and LR.

(b) Establish the order in which the pieces must be inserted in the active rectangle to estimate the profits from using BK_1, BK_2, BK_3, and BK_4.

(c) Establish a selection criterion of the next rectangle R_k to be assigned.

(d) Define sensors that deliver online information about the characteristics of the problem at any instant of the process.

The new specific functions are as follows:

(i) *UnionWithTop*(): a function that selects a piece identical to the last one entered into SB and joins it using a horizontal or vertical combination with the block at the top of SB. The function selects the combination that generates the smallest internal loss. In addition, it returns the number of the selected piece; otherwise, it returns 0 (false).

(ii) *IfPieceRep*(P_1, P_2): a function that acts as a sensor to estimate the average number of times that the available pieces fit in the rectangle R_k. If this average is greater than 2, it performs P_1; otherwise, it performs P_2. It returns the value of the parameter that was executed.

(iii) *IfCorrelation*(P_1, P_2): a function that acts as a sensor that correlates the values of v_i and s_i of the available pieces that fit in R_k. If the pieces are correlated by a value > 0.7, then P_1 is performed; otherwise, P_2 is performed. The correlation index ranges from 0 to 1, with 1 indicating that the variables are completely correlated. The correlation is calculated as follows: covariance / (standard deviation v_i $*$ standard deviation s_i). The function returns the value of the executed parameter.

(iv) *IfBigPiece*(P_1, P_2): a function that acts as a sensor to estimate the size of the available pieces. If at least 50% of the available pieces have an area greater than one-eighth of the plate, then P_1 is performed; otherwise, P_2 is performed. The function returns the value of the executed parameter.

(v) BK_3(): a function that acts as a flag to indicate that, in the next execution of the *Cut* or *Add-p* function

(whichever occurs first), the indicator BK_3 must be used. The function always returns 3 (true).

(vi) BK_4(): a function that acts as a flag to indicate that, in the next execution of the *Cut* or *Add-p* function (whichever occurs first), the indicator BK_4 must be used. The function always returns 4 (true).

(vii) *StopUnion*(): a function that acts as a flag and that can stop the automatic horizontal-vertical joining that occurs in SB when a piece is inserted into the next *Add-p* execution. The function returns 1 (true).

(viii) *DescendingArea*(): a function indicating that the list to be used in the next execution of a BK_i estimator must use the pieces in order from greatest to smallest areas. This function returns 2 (true).

(ix) *AscendingArea*(): a function indicating that the list to be used in the next execution of a BK_i estimator must use the pieces in order from smallest to greatest areas. This function returns 2 (true).

(x) *DescendingProp*(): a function indicating that the list to be used during the next execution of a BK_i estimator must use the pieces in order from longest to shortest length, as long as the length is greater than the width. If not, the piece is considered to be rotated 90 degrees. This function returns 3 (true).

(xi) *AscendingProp*(): a function indicating that the list to be used during the next execution of a BK_i estimator must use the pieces in order from shortest to longest length, as long as the length is greater than the width. If not, the piece is considered to be rotated 90 degrees. This function returns 4 (true).

(xii) *DescendingLength*(): a function indicating that the list to be used during the next execution of a BK_i estimator must use the pieces in order from longest to shortest length. This function returns 5 (true).

(xiii) *DescendingWidth*(): a function indicating that the list to be used during the next execution of a BK_i estimator must use the pieces in order from greatest to smallest width. This function returns 6 (true).

(xiv) *UPDOWN_PROP*(): a function indicating that the list to be used during the next execution of a BK_i estimator must use a decreasing ranking of the pieces for profit versus area v_i/s_i. This is the default value. This function returns 3 (true).

(xv) *MAXWL_WASTE*(): this function acts as a flag, which indicates that, in the next *Cut*() execution, the rectangle R_k with the largest area must be selected. It returns 1 (false) or 2 (true) if there is a rectangle R_k.

2.3. Evolution and Evaluation of the Algorithms. For the evolution and evaluation of algorithms, 46 instances of problem CW_TDC were used (Table 1) [9, 11, 35]. The instances are classified into three groups. The first group (GT1) has 12 instances for the evolution stage, and the second and third groups (GT2 and GT3) are used for the evaluation stage. Both groups have 14 and 20 instances,

```
(1) OP ⟵ ValStandard_v/s;
(2) BKj ⟵ ValStandard_BK1;
(3) Iter ⟵ 0;
(4) While Iter < n Do
(5)    Iter ⟵ Iter + 1;
(6)    Rk ⟵ MinArea(); MINWL_WASTE();
(7)    Var ⟵ Cut (OP, BKj, Rk);
(8)    Var1 ⟵ Add(OP, BKj);
(9) Return Waste();
```

Algorithm 1: CONS algorithm.

respectively. The criterion used to separate them is to divide the total area of the plate by the sum of the areas of all the available pieces (including the ones that are repeated). Each group has the following instances:

(i) GT1: "2s," "Hchl4s," "CHL2s," "CHL5s," "Hchl3s," "OF1," "OF2," "Hchl5s," "A5," "A4," "Hchl6s," and "STS4S."

(ii) GT2: "CHL6," "CHL1s," "APT34," "CHL7," "A3," "Hchl7s," "APT35," "APT36," "APT30," "Hchl2," "CU7," "Hchl1," "A2s," and "APT38."

(iii) GT3: "CU1," "wang1," "APT37," "APT39," "STS2s," "APT33," "APT32," "CU11," "CU9," "CU8," "CU2," "APT31," "CU10," "A1s," "CU4," "W," "3s," "CU6," "CU5," and "CU3."

2.4. The Fitness Function. The fitness function considers two objectives. The first one is the quality of the algorithm or relative error, whereby the smaller the relative error, the greater the quality. The second criterion considers the relative deviation of the algorithm's number of nodes related to an initially fixed number of nodes. Both terms are expressed in Equation (1). The first term is determined by a mathematical function where u_i and z_i are the optimum values for instance i and the value obtained by the algorithm when such an instance is solved. Additionally, n represents the total number of instances, and α is a numerical value used to give a certain priority or importance to the different terms in the fitness function. The second term of the fitness function is the limit of the number of nodes that an algorithm can have, where l_t represents the initial number of predefined nodes and l_a indicates the number of nodes of the generated algorithm. Then, the fitness function f_p is the union of both terms and measures the performance of the algorithms:

$$f_p = \frac{\alpha}{n} \sum_{i=1}^{n} \frac{|u_i - z_i|}{u_i} + (1 - \alpha) \frac{|l_t - l_a|}{l_t}. \quad (1)$$

2.5. Tools and Parameters. Performing the evolutionary process uses an adaptation of the platform originally developed to implement the GP application GPC++ and designed to evolve tree structures [36]. The process was

performed using Windows 7 on a computer with a 2.5 GHz i5 processor and 8 GB RAM.

A population size of 1000 individuals and 100 generations was used, and the crossover and mutation probabilities were set at 85% and 5%, respectively. The "ramped half-and-half" method was used to create the initial population, with a controlled initial tree size that could later grow to a height of 13. The selection of the fittest individual was performed by a tournament. The mutations used were "swap mutation" and "shrink mutation" [32], and the crossover was performed by exchanging tree branches.

3. Results

The experiment is made up of two parts: the evolution process and the evaluation process. In the first process, the new algorithms face the GT1 set of instances, and the experiment is repeated 30 times to select the best algorithm of each execution. With the selected 30 algorithms, the second process follows, which is divided into two parts: First, the algorithms are evaluated with the GT2 set of instances and later, with GT3.

3.1. Convergence. In the 30 executions of the experiment, the convergence curve shows that the individuals of each generation systematically converge until reaching an average error of between 2 and 4%. The graph in Figure 6 shows the convergence of each of the 30 executions considering a population of 1000 individuals per generation and with a total of 100 generations. In general, it is observed that the fitness values for generation 1 begin with values ranging between approximately 104 and 110%, gradually decreasing until the algorithms reach fitness between 14 and 18%. During the first generations, the error is over 100% because the fitness function also considers the number of nodes, and during those generations, there are algorithms that do not assign any piece and that have different number of nodes than in the initial configuration.

3.2. Algorithm Generation. The best resulting algorithm of each of the executions is selected. Table 2 shows the details of the best algorithms found. The first column indicates the name of the best algorithm of the corresponding execution; it is denoted using the letter A followed by a number indicating the number of the execution from which it comes from. The second column indicates the algorithm's average fitness with the 12 instances of GT1. The third column represents the average error. The fourth and fifth columns indicate the best and worst errors found by the algorithm in one of the 12 GT1 instances. The sixth column shows the standard deviation, and the seventh, eighth, and ninth columns show the number of hits (instances where the algorithm finds the optimum solution), the number of nodes, and the height of the algorithm. Finally, the last column shows the computation time each execution required.

Of the 30 selected algorithms, 16 reach optimum values for at least one instance. Consequently, the best error value is 0.00. In general, all the algorithms are capable of determining a near optimum solution for some of the 12 instances. This is evidenced by the best error value of 0.60%. In contrast, an error of 14.85% found as a worst error average shows that all the algorithms face some difficulty with at least one of the instances. The lowest error average is found in execution 12 at approximately 3.96%, and the lowest fitness average is also found in algorithm 12 at 3.49%. The required computer time for evolution is 8.19 minutes for execution 5, and the greatest time is 10.99 minutes for execution 13. The average fitness is 5.22%, and the best average is 5.93%.

3.3. Algorithm Evaluation. The generated algorithms are robust, and they do not over specialize. To demonstrate this, an evaluation process of the best algorithms found was used. This process consists of evaluating the 30 best algorithms in instances different from those used in their creation. Therefore, groups of instances GT2 and GT3 were used. Table 3 shows the evaluation results of the 30 algorithms with instances of group GT2. Considering that the number of instances used is greater and that these instances have more combinations, better results are observed compared to the results in the evolution process. Regarding the lower fitness average, there are seven executions under 4.00%, in contrast to the case of evolution that has only two. The highest fitness average value increased by approximately 2.00%, specifically, from 6.52% to 8.33%. The same effect is observed in the error average. In the column that shows the number of hits, it is observed that no algorithm found an optimum value. The computer time required to solve the 14 instances of group GT2 is between 11.0 and 30.0 seconds for each algorithm.

The produced algorithms present similar computational performances in the evolution and evaluation stages. The instances used to evaluate the algorithms present greater flexibility in terms of the ratio area of the plate/area of the pieces. Table 4 shows the evaluation results of the 30 algorithms with instances of group GT3. An improvement in the results is observed compared to the evolution of the instances of group GT1 and the evaluation with instances of group GT2. The lowest fitness average is 2.14%, while the greatest is 5.37%. As observed in the table, the new algorithms find at least one optimum solution for an instance. However, the computer time required by the algorithms increases.

The size of the algorithms tends to stabilize at the initially predefined size. There is an indirect evolution of the size of an algorithm during its evolutionary construction toward the predefined size, as specified in Equation (1). Specifically, this effect is supervised by the second term of the fitness function. Table 2 shows that the 30 best algorithms found the same number of initially defined nodes. In other words, the evolutionary process converges toward a search region where algorithms of the desired size are found. However, the distribution of nodes in the tree is varied.

Results in Table 2 suggest that is possible to inspect algorithms of a given size by simply fixing the parameter value in the fitness function. Thus, the search for algorithms

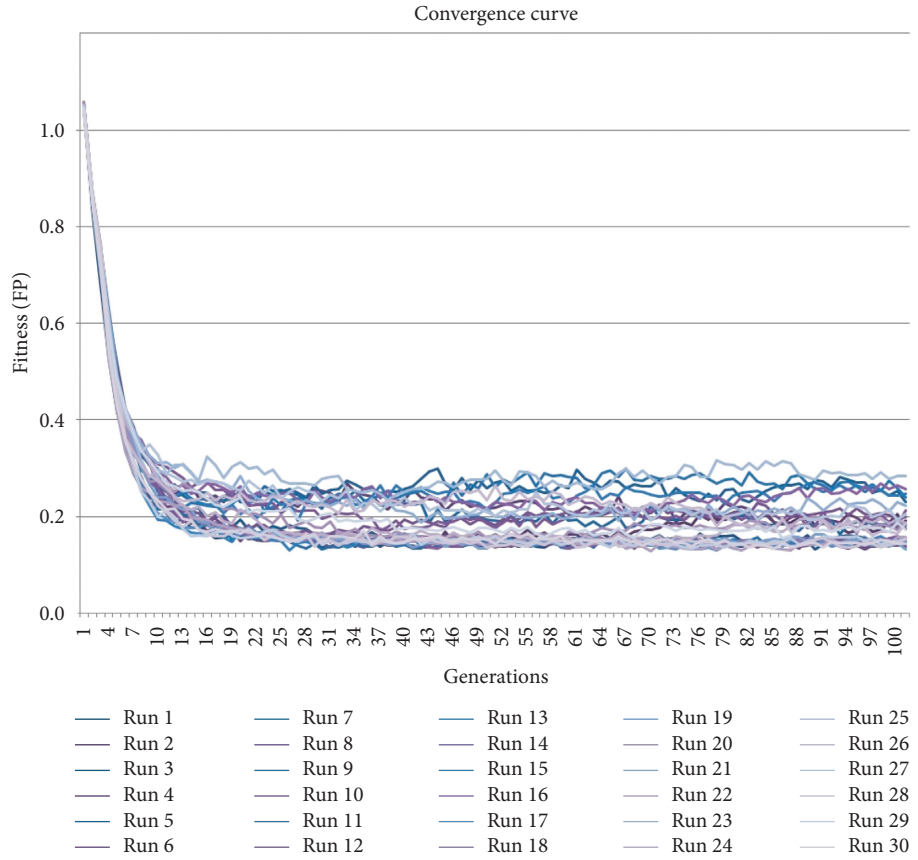

Figure 6: Convergence for the 30 runs.

Table 1: Testing instances.

Number of instances	Instances	Reference
14	OF1, OF2, W, CU1, CU2, CU3, CU4, CU5, CU6, CU7, CU8, CU9, CU10, CU11	Fayard et al. [11]
14	STS2, STS4, A1s, A2s, STS2s, STS4s, CHL1s, CHL2s, A3, A4, A5, CHL5, CHL6, CHL7	Cung et al. [35]
5	Hchl3s, Hchl4s, Hchl5s, Hchl6s, Hchl7s	Álvarez-Valdés et al. [9]
9	APT30, APT31, APT32, APT33, APT34, APT35, APT36, APT37, APT38, APT39	Álvarez-Valdés et al. [9]
4	2s, wang1, wang2, wang3	Others

may focus on desired regions that may be identified beforehand, considering the sizes of the heuristics that already exist for the problem at hand. An example is algorithm A13 in Table 2, with 13 nodes and a height of 4, which has a fitness average of 3.66% for a total of 46 instances where their performance was measured. Figure 7 shows the A13 algorithm's tree representation that stands out in both stages of evolution and evaluation. The main characteristic of A13 is its left branch, which has an instruction *While* that generates a constructive cycle as long as one of the available pieces fits into the plate. The Cut terminal begins when the first piece is assigned, and it divides the initial plate, obtaining two new smaller plates. Then, the Add-p terminal that selects the first or second plate is executed and generates the unions based on an estimator *BKi*. The constructive cycle repeats until it meets the finishing criteria.

The found algorithms follow a constructive and an improvement logic. All the generated algorithms have at least one cycle, and within them, they build solutions from an initial plate until no other piece fits in the plate. The algorithms represented in Figure 8 were selected because they were among the five best algorithms of the experiment. Algorithms A4 and A14 have a *while* cycle composed of a set of functions and terminals, among which Add-p and Cut are

Table 2: Best algorithm of each of the 30 runs.

Algorithm	Avg. fitness (%)	Avg. error (%)	Best error (%)	Worst error (%)	SD (%)	No. of hits	No. of nodes	Tree height	Time (s)
A1	6.52	7.40	0.00	15.22	5.17	1	13	4	553.00
A2	4.69	5.33	1.36	20.75	5.42	0	13	4	560.82
A3	5.84	6.63	0.00	14.37	4.36	1	13	3	592.03
A4	4.38	4.96	2.02	18.29	4.51	0	13	3	532.11
A5	5.18	5.89	0.86	10.57	3.67	0	13	5	491.15
A6	5.85	6.66	1.36	12.93	3.73	0	13	3	595.45
A7	4.28	4.85	1.36	10.74	2.91	0	13	4	564.03
A8	5.85	6.66	1.36	12.93	3.73	0	13	4	599.26
A9	5.84	6.63	0.00	14.37	4.36	1	13	4	603.54
A10	4.63	5.26	0.00	11.64	3.79	1	13	4	600.85
A11	5.19	5.91	1.36	14.96	3.65	0	13	3	511.67
A12	3.49	3.96	0.00	9.83	2.63	1	13	3	556.78
A13	4.50	5.14	0.00	13.87	3.98	1	13	4	659.11
A14	3.92	4.45	0.75	10.23	3.09	0	13	4	597.02
A15	5.63	6.41	1.31	20.75	6.47	0	13	6	596.93
A16	4.64	5.28	0.75	13.17	3.40	0	13	4	518.51
A17	6.15	6.98	0.86	17.77	5.19	0	13	3	607.99
A18	5.84	6.63	0.00	14.37	4.36	1	13	3	607.98
A19	6.34	7.20	0.00	14.82	4.62	1	13	4	591.04
A20	5.03	5.70	0.00	12.52	4.02	1	13	5	618.95
A21	5.84	6.63	0.00	14.37	4.36	1	13	4	596.01
A22	5.59	6.37	1.36	13.14	4.16	0	13	5	592.75
A23	5.09	5.81	0.00	17.95	4.85	1	13	4	598.61
A24	5.84	6.63	0.00	14.37	4.36	1	13	4	580.30
A25	4.23	4.80	0.00	14.48	4.26	1	13	5	648.99
A26	5.15	5.86	0.00	17.95	4.62	1	13	4	516.53
A27	5.84	6.63	0.00	14.37	4.36	1	13	4	602.66
A28	4.38	4.96	2.02	18.29	4.51	0	13	4	587.45
A29	4.69	5.33	1.36	20.75	5.42	0	13	5	549.60
A30	6.25	7.11	0.00	15.81	4.82	1	13	4	573.66
Average	5.22	5.93	0.60	14.85	4.29	0.5	13	4	580.16

Table 3: Evaluation of algorithms with instances from group GT2.

Algorithm	Avg. fitness (%)	Avg. error (%)	Best error (%)	Worst error (%)	SD (%)	Hits	Time (s)
A1	8.33	9.35	2.34	16.12	4.43	0	21.0
A2	3.53	3.97	1.47	8.14	2.20	0	20.0
A3	6.04	6.80	3.08	16.12	3.46	0	18.0
A4	3.51	3.94	0.88	7.24	2.43	0	17.0
A5	4.43	4.95	1.41	13.89	3.54	0	25.0
A6	5.64	6.34	1.32	18.53	4.61	0	23.0
A7	4.53	5.09	1.01	11.76	2.85	0	15.0
A8	5.64	6.34	1.32	18.53	4.61	0	14.0
A9	6.04	6.80	3.08	16.12	3.46	0	17.0
A10	7.15	8.03	3.09	16.12	4.11	0	15.0
A11	4.29	4.83	1.32	11.76	2.94	0	13.0
A12	4.56	5.12	1.07	18.53	4.57	0	11.0
A13	4.32	4.84	1.38	12.20	3.24	0	19.0
A14	3.61	4.06	1.59	8.14	2.08	0	20.0
A15	3.55	4.00	1.21	12.20	3.26	0	22.0
A16	4.29	4.83	1.32	11.76	2.94	0	27.0
A17	6.00	6.73	1.64	18.50	4.71	0	21.0
A18	6.04	6.80	3.08	16.12	3.46	0	22.0
A19	8.33	9.35	2.34	16.12	4.43	0	27.0
A20	5.73	6.46	1.64	16.12	4.03	0	26.0
A21	6.04	6.80	3.08	16.12	3.46	0	20.0
A22	6.30	7.07	1.64	18.50	4.70	0	19.0
A23	3.89	4.38	1.83	8.97	2.18	0	20.0
A24	6.04	6.80	3.08	16.12	3.46	0	24.0

Table 3: Continued.

Algorithm	Avg. fitness (%)	Avg. error (%)	Best error (%)	Worst error (%)	SD (%)	Hits	Time (s)
A25	5.13	5.78	2.00	16.12	3.55	0	15.0
A26	6.06	6.81	3.08	16.12	3.38	0	25.0
A27	6.04	6.80	3.08	16.12	3.46	0	19.0
A28	3.51	3.94	0.88	7.24	2.43	0	17.0
A29	3.53	3.97	1.47	8.14	2.20	0	14.0
A30	5.99	6.73	1.32	18.53	4.94	0	15.0
Average	5.27	5.92	1.90	14.20	3.50	0	19.0

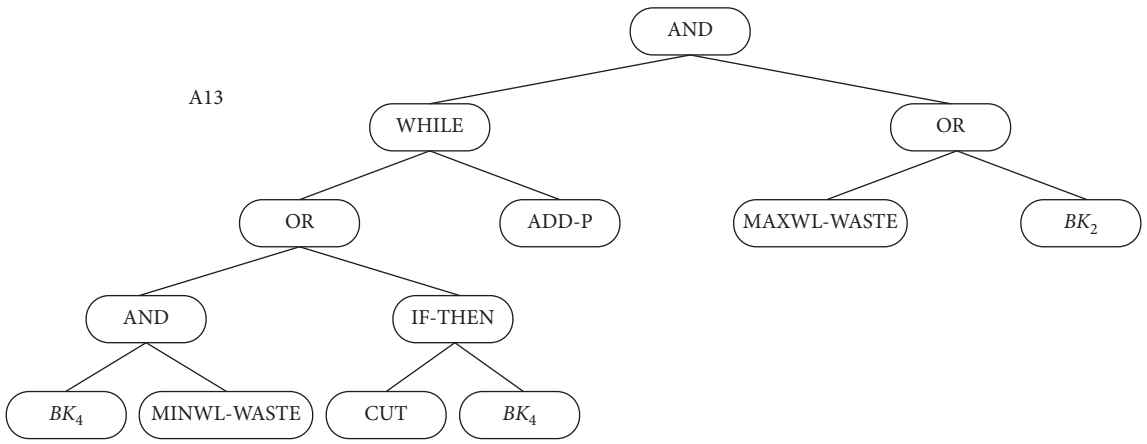

Figure 7: Tree of the algorithm from run 13.

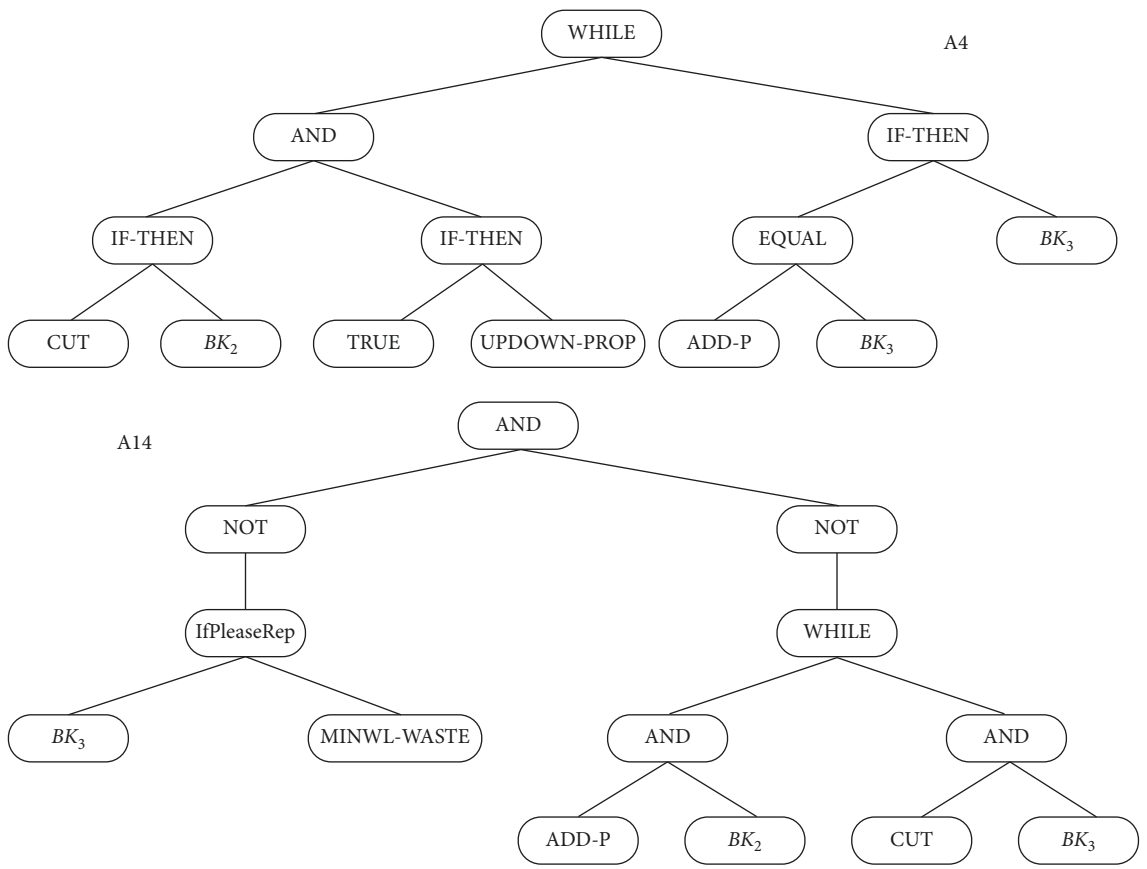

Figure 8: Tree instructions for algorithms A4, A13, and A14.

Table 4: Evaluation of algorithms with instances from group GT3.

Algorithm	Avg. fitness (%)	Avg. error (%)	Best error (%)	Worst error (%)	SD (%)	No. of hits	Time (s)
A1	5.09	5.71	0	12.86	3.91	1	48.0
A2	2.32	2.60	0	5.56	1.33	1	44.0
A3	4.22	4.74	0	8.96	2.93	1	51.0
A4	3.33	3.74	0	11.58	2.61	1	41.0
A5	4.03	4.52	0	10.12	2.74	1	44.0
A6	3.02	3.39	0	8.23	2.31	1	46.0
A7	3.41	3.83	0	8.34	2.14	1	45.0
A8	3.02	3.39	0	8.23	2.31	1	42.0
A9	4.22	4.74	0	8.96	2.93	1	40.0
A10	4.21	4.73	0	11.85	3.13	1	39.0
A11	2.66	2.99	0	6.96	1.82	1	50.0
A12	3.70	4.15	0	20.42	4.41	1	49.0
A13	2.14	2.41	0	7.80	1.74	1	48.0
A14	2.68	3.01	0	8.23	2.09	1	41.0
A15	2.31	2.59	0	7.80	2.01	1	43.0
A16	2.66	2.99	0	6.96	1.82	1	42.0
A17	5.37	6.03	0	15.73	4.14	1	51.0
A18	4.22	4.74	0	8.96	2.93	1	42.0
A19	5.09	5.71	0	12.86	3.91	1	43.0
A20	4.43	4.97	0	13.20	3.43	1	38.0
A21	4.22	4.74	0	8.96	2.93	1	45.0
A22	5.37	6.03	0	15.73	4.14	1	42.0
A23	2.43	2.73	0	6.96	1.76	1	45.0
A24	4.22	4.74	0	8.96	2.93	1	48.0
A25	3.52	3.96	0	8.96	2.80	1	43.0
A26	4.22	4.74	0	8.96	2.93	1	50.0
A27	4.22	4.74	0	8.96	2.93	1	51.0
A28	3.33	3.74	0	11.58	2.61	1	51.0
A29	2.32	2.60	0	5.56	1.33	1	42.0
A30	3.02	3.39	0	8.23	2.31	1	46.0
Average	3.63	4.08	0	9.89	2.71	1	45.0

focused on the logical constructive steps. Add-p is in charge of joining blocks and Cut and of assigning them to the plate, repeating this process until the plate is completed.

The algorithms assemble the location of the pieces using the criteria of best fit. This behavior is found specifically in the BK_i estimators because each estimator has a different criteria to fit pieces. If the default BK_1 estimator is not useful for fitting pieces, then it is possible for the algorithm to use a different estimator in one of its tree's branches. Moreover, there is another way to use the best piece, based on the ordered lists that offer the best possible fit, using different methods of sorting the pieces during the selection. These methods may be from largest to smallest or from smallest to largest, considering its own criterion that indicates the type of sorting applied (width, length, area, etc.). A clear example is algorithm A4 of Figure 8, which has a Cut terminal in its left branch with a BK_2 estimator and an Add-p terminal and two BK_3 terminals in its right branch.

The found algorithms are a generalization of good existing heuristics for this problem. The CONS heuristic is the fundamental base of the generated algorithms. The algorithms find a solution that begins with the greatest loss and gradually decreases as pieces are assigned to the solution. Figure 8 provides two examples that show that the Add-p and Cut terminals, which are part of the CONS heuristic, the base for the algorithms to solve the instances. Both algorithms show such terminals, and they are generally preceded by the function *while*, which produces repetition a number of times until the algorithms reach the optimum or nearly optimum result.

Constructive cycles prevail in the found algorithms. In most of the analyzed algorithms, there are cycles that try to find a possible solution using estimators and piece lists. Always connected by a While cycle, Add-p and Cut prioritize the construction of blocks to find the solution, as can be observed in the right branch of algorithm A14 presented in Figure 8. This branch, with only seven nodes, executes a number of combinations to solve the problem. Because the While is the base of the branch, the algorithm ensures that the execution is repeated in the other six nodes until the stop criterion is reached. There are two And functions in the other six nodes, leaving the last four nodes (Add-p with BK_2 and Cut with BK_3 with While) as the base of the constructive cycles.

The cycles in the algorithms operate as instruction compacters. The resulting algorithms are capable of repeating the process a great number of times, which are not always the same, using few code lines. In this way, a great number of operations are conducted, but instructions are compacted in small and easy to understand branches. An example appears in Figure 8, where the three algorithms

Table 5: Evaluation of algorithms with instances in groups GT2 and GT3.

Algorithm	Avg. error (%) GT2	Hits GT2	Time (s) GT2	Avg. error (%) GT3	Hits GT3	Time (s) GT3
CONS	4.53	0	2.45	3.36	0	2.75
GRASP	1.63	2	142.44	0.73	7	158.4
TABU	0.34	5	1459.86	0.25	9	1730.1

have 13 nodes each and are able to solve instances that involve from 10 pieces to over 50 pieces without the need to use more than one While cycle in their structure.

Several algorithms obtained are competitive with respect to a state-of-the-art constructive algorithm. Table 5 shows a summary of the results obtained by CONS, GRASP, and TABU algorithms presented in Álvarez-Valdés et al. [9] for instances GT2 and GT3. It is observed that for "Avg. error (%)," the four evolved constructive algorithms A2, A14, A15, and A29 have a lower value than the value generated by CONS algorithm for both groups of data GT2 and GT3. Additionally, algorithms A4, A23, and A28 have a lower value only with instances in GT2; meanwhile, algorithms A6, A8, A11, A13, A16, A23, and A30 have better performance with instances in GT3. Also, the new algorithms find at least one optimal solution for instances in GT3 compared with none optimal solution found by CONS. GRASP and TABU algorithms are more effective since they obtain a lower average error. With respect to the running time required by CONS, it is lower than the average of the running time required by the new algorithms.

4. Conclusion

This paper describes a computational model and experiment that allowed for the generation of algorithms to solve a set of instances of the guillotine-cutting problem. The generated algorithms were decoded from tree structures that were evolved with a computational tool based on GP. The functions that constitute the basic components of the produced algorithms were deduced by identifying the basic components of an existing algorithm. Other functions inspired by the geometric and algorithmic solutions of the problem were added to provide greater variability in the algorithm search. The best 30 algorithms were identified and tested with 46 representative instances of the problem. The average error of the algorithms varied between 3.00 and 5.00%. The generated algorithms are able to find better results by working on instances with more possible combinations among their pieces. The computational results are similar between instances of different combinatory degrees.

Data Availability

The data set used in this paper is very common in the field of cutting problems and can be obtained from a public web site: http://people.brunel.ac.uk/~mastjjb/jeb/info.html.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research was partially funded by the Complex Engineering Systems Institute (ICM-FIC: P05-004-F, CONICYT: FB0816).

References

[1] R. N. Morabito and L. Belluzzo, "Optimizing the cutting of wood fibre plates in the hardboard industry," *European Journal of Operational Research*, vol. 183, no. 3, pp. 1405–1420, 2007.

[2] S. C. Poltroniere, K. C. Poldi, F. M. B. Toledo, and M. N. Arenales, "A coupling cutting stock-lot sizing problem in the paper industry," *Annals of Operations Research*, vol. 157, no. 1, pp. 91–104, 2008.

[3] K. Matsumoto, S. Umetani, and H. Nagamochi, "On the one-dimensional stock cutting problem in the paper tube industry," *Journal of Scheduling*, vol. 14, no. 3, pp. 281–290, 2011.

[4] E. Malaguti, R. Durán, and P. Toth, "Approaches to real world two-dimensional cutting problems," *Omega*, vol. 47, pp. 99–115, 2014.

[5] O. Oliveira, D. Gamboa, and P. Fernandes, "An information system for the furniture industry to optimize the cutting process and the waste generated," *Procedia Computer Science*, vol. 100, pp. 711–716, 2016.

[6] K.-T. Park, J.-H. Ryu, H.-K. Lee, and I.-B. Lee, "Development of a heuristic algorithm for cutting stock problems in flat glass production processes," *Journal of Chemical Engineering of Japan*, vol. 45, no. 3, pp. 219–227, 2012.

[7] G. Wäscher, H. HauBner, and H. Schumann, "An improved typology of cutting and packing problems," *European Journal of Operational Research*, vol. 183, no. 3, pp. 1109–1130, 2007.

[8] M. R. Garey and D. S. Johnson, *Computers and Intractability. A Guide to the Theory of NP-Completeness: A Series of Books in the Mathematical Sciences*, WH Freeman and Company, San Francisco, CA, USA, 1979.

[9] R. Álvarez-Valdés, A. Parajon, and J. M. Tamarit, "A tabu search algorithm for large-scale guillotine (un)constrained two-dimensional cutting problems," *Computers & Operations Research*, vol. 29, no. 7, pp. 925–947, 2002.

[10] R. N. Morabito and V. Pureza, "A heuristic approach based on dynamic programming and and/or-graph search for the constrained two-dimensional guillotine cutting problem," *Annals of Operations Research*, vol. 179, no. 1, pp. 297–315, 2010.

[11] D. Fayard, M. Hifi, and V. Zissimopoulos, "An efficient approach for large-scale two-dimensional guillotine cutting stock problems," *Journal of the Operational Research Society*, vol. 49, no. 12, pp. 1270–1277, 1998.

[12] X. Song, C. B. Chu, R. Lewis, Y. Y. Nie, and J. Thompson, "A worst case analysis of a dynamic programming-based heuristic algorithm for 2D unconstrained guillotine cutting," *European Journal of Operational Research*, vol. 202, no. 2, pp. 368–378, 2010.

[13] P. C. Gilmore and R. E. Gomory, "A linear programming approach to the cutting stock problem-Part II," *Operations Research*, vol. 11, no. 6, pp. 863–888, 1963.

[14] A. I. Hinxman, "The trim-loss and assortment problems: a survey," *European Journal of Operational Research*, vol. 5, no. 1, pp. 8–18, 1980.

[15] N. Christofides and C. Whitlock, "An algorithm for two-dimensional cutting problems," *Operations Research*, vol. 25, no. 1, pp. 30–44, 1977.
[16] P. Y. Wang, "Two algorithms for constrained two-dimensional cutting stock problems," *Operations Research*, vol. 31, no. 3, pp. 573–586, 1983.
[17] J. C. Herz, "Recursive computational procedure for two-dimensional stock cutting," *IBM Journal of Research and Development*, vol. 16, no. 5, pp. 462–469, 1972.
[18] M. Hifi and V. Zissimopoulos, "Constrained two-dimensional cutting: an improvement of Christofides and Whitlock's exact algorithm," *Journal of the Operational Research Society*, vol. 48, no. 3, pp. 324–331, 1997.
[19] F. Vasko, "A computational improvement to Wang's two-dimensional cutting stock algorithm," *Computers & Industrial Engineering*, vol. 16, no. 1, pp. 109–115, 1989.
[20] Y. Cui and X. Zhang, "Two-stage general block patterns for the two-dimensional cutting problem," *Computers & Operations Research*, vol. 34, no. 10, pp. 2882–2893, 2007.
[21] A. Lodi, M. Monaci, and E. Pietrobuoni, "Partial enumeration algorithms for two-dimensional bin packing problem with guillotine constraints," *Discrete Applied Mathematics*, vol. 217, pp. 40–47, 2017.
[22] N. Christofides and E. Hadjiconstantinou, "An exact algorithm for orthogonal 2-D cutting problems using guillotine cuts," *European Journal of Operational Research*, vol. 83, no. 1, pp. 21–38, 1995.
[23] J. E. Beasley, "Algorithms for unconstrained two-dimensional guillotine cutting," *Journal of the Operational Research Society*, vol. 36, no. 4, pp. 297–306, 1985.
[24] F. G. Cintra, F. K. Miyazawa, Y. Wakabayashi, and E. C. Xavier, "Algorithms for two-dimensional cutting stock and strip packing problems using dynamic programming and column generation," *European Journal of Operational Research*, vol. 191, no. 1, pp. 61–85, 2008.
[25] Y. Chen, "A recursive algorithm for constrained two-dimensional cutting problems," *Computational Optimization and Applications*, vol. 41, no. 3, pp. 337–348, 2007.
[26] E. Talbi, *Metaheuristics: from Design to Implementation*, John Wiley and Sons, Hoboken, NJ, USA, 2009.
[27] R. N. Morabito, M. N. Arenales, and V. F. Arcaro, "An and-or-graph approach for two-dimensional cutting problems," *European Journal of Operational Research*, vol. 58, no. 2, pp. 263–271, 1992.
[28] M. Hifi, "The DH/KD algorithm: a hybrid approach for unconstrained two-dimensional cutting problems," *European Journal of Operational Research*, vol. 97, no. 1, pp. 41–52, 1997.
[29] V. Parada, R. Palma, D. Sales, and A. Gomes, "A comparative numerical analysis for the guillotine two-dimensional cutting problem," *Annals of Operations Research*, vol. 96, no. 1–4, pp. 245–254, 2000.
[30] J. R. Koza, M. A. Keane, M. J. Streeter, W. Mydlowec, J. Yu, and G. Lanza, *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*, Kluwer Academic Publishers, Norwell, MA, USA, 2005.
[31] J. R. Koza, "Human-competitive results produced by genetic programming," *Genetic Programming and Evolvable Machines*, vol. 11, no. 3-4, pp. 251–284, 2010.
[32] R. Poli, W. B. Langdon, and N. F. McPhee, *A Field Guide to Genetic Programming*, Lulu Enterprises UK Ltd., London, UK, 2008.
[33] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*, John Wiley & Sons, Bologna, Italy, 1990.
[34] E. G. Coffman Jr., M. R. Garey, and D. S. Johnson, "Approximation algorithms for bin packing: a survey," in *Approximation Algorithms for NP-Hard Problems*, D. S. Hochbaum, Ed., pp. 46–93, PWS Publishing Co., Boston, MA, USA, 1997.
[35] V. Cung, M. Hifi, and B. Le Cun, "Constrained two-dimensional cutting stock problems a best-first branch-and-bound algorithm," *International Transactions in Operational Research*, vol. 7, no. 3, pp. 185–210, 2000.
[36] A. Fraser and T. Weinbrenner, "Genetic programming C++ class library," 1997, http://www0.cs.ucl.ac.uk/staff/ucacbbl/ftp/weinbenner/gp.html.