On the Concept of Density Control and its Application to a Hybrid Optimization Framework: Investigation into Cutting Problems

Alexei B. de Aguiar¹, Plácido R. Pinheiro²∗, André L. V. Coelho²∗∗, Tibérius O. Bonates²∗∗

¹University of Fortaleza, Graduate Program in Applied Informatics
Av. Washington Soares 1321/J30, 60811-905, Fortaleza, CE, Brazil

Abstract

The Generate-and-Solve (GS) methodology is a hybrid method that combines a metheuristic component with an exact solver. GS has been recently introduced in the literature in order to solve cutting and packing problems, showing promising results. The GS framework includes a metheuristic engine (e.g., a genetic algorithm) that works as a generator of reduced instances of the original optimization problem, which are, in turn, formulated as a mathematical programming problems and solved by an integer programming solver. In this paper, we present an extended version of GS focusing primarily on the concept of a new Density Control Operator (DCO). The role of this operator is to adaptively control the dimension of the reduced instances in such a way as to allow a much steadier progress towards a better solution, thereby avoiding premature convergence. In order to assess the potential of this novel version of the GS methodology, we have conducted computational experiments on a set of difficult benchmark instances of the constrained non-guillotine cutting problem. The results achieved are quantitatively and qualitatively discussed in terms of effectiveness and efficiency, showing that the proposed variant of the GS hybridization framework is highly suitable when effectiveness is a major requirement.

Keywords: Combinatorial optimization, Hybrid metaheuristics, Cutting and packing, Genetic algorithms, Integer linear programming, Density control operator

1. Introduction

Mathematical programming and metheuristic techniques have been intensively investigated in the last decades, with each class showing pros and cons when dealing with hard combinatorial optimization problems [1]. It is well-known that, whenever optimality is a top concern, an exact method is the procedure of choice. However, as the size and complexity of the optimization problems that need to be solved increases, the direct application of exact algorithms becomes prohibitive. Therefore, there is a permanent demand for the development of algorithms that can find acceptable solutions within a reasonable amount of time. In such cases, making use of a heuristic component within a solution framework may be helpful, since flexibility and computational tractability are properties inherent to such methods [2].

Exact and metheuristic techniques typically represent complementary perspectives over the problem-solving process of combinatorial optimization problems. Recently it has become evident that a skilled combination of their ingredients into a single conceptual framework can be a promising strategy, mainly when complex optimization tasks are at hand. In fact, an alternative class of optimization approaches that combine exact and heuristic algorithms, or simply different heuristics algorithms, has emerged and gained increased attention recently. This class of approaches has been referred to by the general label of “hybrid metheuristics” [3, 4, 5].
In this paper we propose the use of a novel genetic operator, called a “Density Control Operator” (DCO) within the recently introduced Generate-and-Solve (GS) hybrid framework. The new operator is shown to improve the efficiency and efficacy of GS over as set of test problems.

The paper is organized as follows. In Section 2 we provide a brief account of some recent advances in the field of hybrid metaheuristics, while in Section 3 we review the main concepts of the Generate-and-Solve framework. In Section 4 we present an integer linear programming (ILP) formulation of the particular cutting problem investigated here. In Section 5 we introduce the novel Density Control Operator, and in Section 6 we discuss its control parameters. In Section 7 we report and discuss our computational results on benchmark instances, in terms of effectiveness and efficiency. Section 8 concludes the paper, while providing some suggestions for future work.

2. Overview of Hybrid Metaheuristic Approaches

Given the distinct and somewhat complementary points of view of exact and heuristic search techniques on optimization problems, it appears natural to try to combine them into more powerful algorithms [6]. Nevertheless, only rather recently this form of hybridization has been more deeply investigated. Indeed, over the last years the interest in hybrid metaheuristics has risen considerably among researchers in combinatorial optimization, as can be seen by the diversity of works on this topic in the recent literature [3, 4, 5, 6, 7].

In [8], for instance, a hybrid approach that combines simulated annealing with local search heuristics was introduced for solving the traveling salesman problem. In [9] the authors presented a hybrid method that applies simulated annealing to improve the population obtained by a genetic algorithm. In [10] a local search algorithm, which utilizes problem-specific knowledge, is incorporated into the genetic operators of a GA instance to solve the multi-constraint knapsack problem. In [11] various kinds of local search methods are embedded into an evolutionary algorithm for locally improving candidate solutions.

Many hybrid algorithms aim at providing optimal solutions in a shorter time, while others focus primarily on getting better heuristic solutions [7]. For instance, in order to reduce the search space, the authors of [12] combine tabu search with an exact method for solving the 0-1 multidimensional knapsack problem. By other means, in [13] a two-phase hybrid method is proposed in which high quality tours for the traveling salesman problem are generated, while the subproblem induced by the set of previous tours is solved exactly on a restricted graph. In [14] a genetic algorithm is combined with a local optimization procedure based on the Coulomb potential technique in order to solve blank packing problems. The hybrid algorithm is shown to provide significant improvement over a genetic algorithm when solving large instances. Other promising ideas include local branching [15], in which a general-purpose commercial solver is used to effectively explore solution subspaces that are defined and controlled at a strategic level by a simple external branching framework.

Combinations of different metaheuristics, or of metaheuristics with exact methods (such as mathematical programming), have given rise to different hybridization schemes. In [4] a taxonomy of hybrid metaheuristic components was presented that embraces those different schemes by distinguishing them into two levels. In the low-level scheme, the result is a functional composition of a single optimization method. In that class, a specific step of a metaheuristic is replaced by the application of another metaheuristic. In the high-level scheme, the different algorithms are self-contained, with none of them having direct influence over the internal workings of the other algorithms.

In [7] an alternative taxonomy of existing approaches combining exact and metaheuristic algorithms for combinatorial optimization was presented, whereby the following two main categories of hybrid approaches are distinguished: collaborative and integrative. By collaboration, it is meant that the constituent algorithms exchange information to each other, but none is part of the other. The algorithms may execute sequentially, intertwined, or in parallel. By integration, it is meant that one technique is a subordinate component of the other. Thus, there is a clear distinction between a master algorithm, which can be either an exact or a metaheuristic algorithm, and a slave algorithm. Yet another classification of hybrid methods can be found in [5].

3. Generate-and-Solve

A hybrid methodology called Generate-and-Solve (GS) has been recently introduced in the literature [16, 17] with the aim of decomposing the original problem into two conceptual levels. In the GS framework an exact method
Figure 1: The GS hybrid framework.

(encapsulated in the Solver of Reduced Instances (SRI) component) works with reduced instances of the original problem (i.e. subproblems) that still preserve the original problem’s conceptual structure (see Figure 1). Thus, an optimal solution to a given subproblem will also be a feasible solution to the original problem. At a different level, a metaheuristic component of the framework works on a complementary optimization problem, namely the design of reduced instances of the original problem formulated as mathematical programming models (in our case, integer linear programming models).

This metaheuristic component is referred to as the Generator of Reduced Instances (GRI), whose goal is to determine the subset of points of the reducible structure that could derive the best subproblem instance, i.e., the subproblem which, when submitted to the SRI, would bring about the feasible solution with the highest possible objective function value. In this scenario, the objective function values of the solutions that could be in fact realized by the solver are used as figure of merit (fitness) of their associated subproblems, thus guiding the metaheuristic search process. The interaction between GRI and SRI is iterative and proceeds until a certain stopping condition is satisfied.

The metaheuristic chosen to implement the Generator of Reduced Instances has been a Genetic Algorithm (GA) [18]. This option is due mainly to the good levels of flexibility and adaptability exhibited by this class of evolutionary algorithms when dealing with a wide range of optimization problems [19]. The genetic representation of the individuals (chromosomes) follows a binary encoding that indicates which decision variables belonging to the reducible structure will be kept in the new subproblem to be generated. In other words, those genes having ‘1’ as alleles define the subset of variables that generates the reduced instance. The method of choice for the Solver of Reduced Instances is the ILOG CPLEX solver [20], a state-of-the-art integer linear programming solver which implements algorithms such as branch-and-bound and branch-and-cut [21] for the solution of ILP problems. The solver library also incorporates sets of strategies, heuristics, and problem reduction techniques that complement the main exact method in a way as to enhance its overall performance.

According to the classification of hybrid optimization models proposed in [7], the GS methodology falls into the category of integrative combinations. The quality of the solutions to the instances generated by the metaheuristic is determined when the subproblems are solved by the SRI, and the best solution obtained throughout the whole metaheuristic process is deemed to be the final solution to the original problem. However, there are cases where it is not possible to solve, or even generate, the reduced problem instance, due to its complexity and/or to other computational limitations of the solver environment. Thus, roughly speaking, the hybrid framework operates with the pragmatic perspective of providing effective solutions to complex optimization tasks by searching for derived subproblems that can still be directly handled by the exact methods one currently has at his/her disposal.

Although yielding high-quality results for some problems studied in the domain of cutting and packing [16, 17], the original version of GS is hampered by a serious drawback, namely, its tendency to produce subproblems of increasing complexity – closer to the original problem – too prematurely, thus defeating the purpose of the hybridization. This drawback is circumvented with the adoption of the Density Control (DCO) Operator mechanism proposed in this paper, whose role is to adaptively control the increase in the size of the generated subproblems so as to allow a much
steadier progress towards a better solution. As a means to assess the potential of the novel version of GS in terms of effectiveness (objective function value) and efficiency (time elapsed) criteria, some experiments have been conducted targeting complicated benchmark instances of constrained non-guillotine cutting. The results achieved by means of these experiments are quantitatively and qualitatively discussed, showing the suitability of the improved hybrid framework.

4. Problem Formulation

In this section we define the optimization problem that will be used in this paper for the purpose of assessing the performance of our variant of the GS algorithm. Our presentation follows that of [22].

The general version of the cutting stock problem involves finding the best allocation of a number of items within a larger region [23]. The problem belongs to the family of NP-complete problems and can be encountered in numerous domains, such as computer science, industrial engineering, logistics, manufacturing, among others. In this paper we are concerned with a variant of the general problem, called the “constrained non-guillotine cutting problem”.

In order to formulate the constrained non-guillotine cutting problem, we resort to the ILP model proposed in [22].

Consider a set of items grouped into \( m \) types. Let us define for each item type \( i \), its length \( l_i \), width \( w_i \), and value \( v_i \). Let us also define an associated limit \( b_i \) on the number of items of a given type that can be produced. Consider also a large object \( B \) that has \( L \) and \( W \) as its length and width dimensions, respectively. A set of items should be obtained by means of orthogonal cuts to this large object. We are looking for a way to cut items that maximizes the total value, obtained as the sum of values of all individual items produced.

We will assume that all items have a fixed orientation, as well as the large object \( B \). For the purpose of distinguishing between the different ways in which an item can be cut from \( B \), we will use the upper left corner of each item as the reference point for deciding where an item will be cut from \( B \).

In what follows we define the decisions variables used in the ILP formulation of the problem. Let \( u_{ide} \) be a binary decision variable that alludes to the decision of whether or not to cut an item of type \( i \) at the coordinate \((d, e)\).

\[
u_{ide} = \begin{cases} 1, & \text{if a type } i \text{ item is cut in position } (d, e) \\ 0, & \text{otherwise} \end{cases}
\]

(1)

The elements \( d \) and \( e \) belong, respectively, to the following discretization sets:

\[
X = \{ x \mid x = \sum_{i=1}^{n} \alpha_i l_i, x \leq L - \min_{1 \leq i \leq m} \{ l_i \}, \alpha_i \in \mathbb{Z}^+ \}
\]

(2)

\[
Y = \{ y \mid y = \sum_{i=1}^{m} \beta_i w_i, y \leq W - \min_{1 \leq i \leq m} \{ w_i \}, \beta_i \in \mathbb{Z}^+ \}. 
\]

(3)

To avoid box interpositions, the incidence matrix \( g_{idepq} \) is defined as:

\[
g_{idepq} = \begin{cases} 1, & \text{if } d \leq p \leq d + l_i - 1 \text{ and } e \leq q \leq e + w_i - 1 \\ 0, & \text{otherwise} \end{cases}
\]

(4)

which has to be computed a priori for each box type \( i \) (\( 1 \leq i \leq m \)), for each coordinate \((d, e)\), and for each coordinate \((p, q)\).

The constrained non-guillotine cutting problem can be formulated as:

\[
\max \sum_{i=1}^{m} \sum_{d \in X} \sum_{e \in Y} v_i u_{ide}
\]

subject to \( \sum_{i=1}^{m} \sum_{d \in X} \sum_{e \in Y} g_{idepq} u_{ide} \leq 1, \forall p \in X, \forall q \in Y \)

(5)

(6)
\[
\sum_{d \in X} \sum_{e \in Y} u_{ide} \leq b_i, \quad 1 \leq i \leq m
\] (7)

\[u_{ide} \in \{0, 1\}, \quad 1 \leq i \leq m, \quad \forall d \in X, \forall e \in Y\] (8)

5. The Density Control Operator

As mentioned in Section 1, a drawback that has limited the effectiveness of GS as originally conceived relates to its propensity for bringing about a rapid increase over the size of the individuals (i.e., reduced instances of the original problem) produced by the GRI component. In other words, the hybrid methodology is hampered by the problem of “uncontrolled density explosion.” We refer to the density of an individual as the ratio between the number of its genes having ‘1’ as allele (hereafter referred to as an “activated gene”) and its total length.

An increase in the density of the individuals of the population tends to generate subproblems that are closer to the original problem, thus possibly yielding better solutions. Although expected, this phenomenon may have an undesirable side-effect if it occurs prematurely. This is because, usually, high densities imply higher complexity to be dealt with by the SRI, which indirectly affects the search process conducted by the GRI, since the time spent in each generation tends to become progressively higher. This may cause a drastic limitation over the number of search iterations performed by the framework, hindering both the effectiveness and efficiency of the whole optimization process.

In order to overcome this deficiency, we have devised a new mechanism, named the Density Control Operator (DCO), that is introduced into the GRI and called after the application of the recombination (crossover) operator. By adopting the DCO, more feedback information from the SRI to the GRI is allowed, in addition to the objective function values of the solutions, as in the original version of the framework. In a given GRI individual, each activated gene, which represents a variable effectively used in the solution generated by the SRI, a credit value is assigned that will be taken into account by the DCO operator. This credit value amounts to the objective function value of the SRI solution. Non-activated genes receive no credit. The credit of each activated gene is used by the DCO to select genes that will be disabled in order to bring the density of the individual closer to a desirable value.

The main parameter related to DCO is the “ideal density” (IDD) (see Subsection 6.2.2), which should be calibrated beforehand. During the application of the Density Control Operator, the following analysis is conducted over each GRI individual \(c_i\). If the density of \(c_i\) is lower than or equal to IDD, nothing is done. Otherwise, a subset of the genes of \(c_i\) having null credit is randomly chosen to be deactivated (i.e., zeroed), as shown in the Density Reduction Algorithm (Algorithm 1). After this stage, if the current density value of \(c_i\) is still greater than IDD, other genes with some associated (non-null) credit will also suffer deactivation, with those having smaller credit values being deactivated first. This last procedure is optional and its execution is controlled by another parameter, PNZCG, described in Subsection 6.2.3.

Below we present a pseudocode of the Density Reduction Algorithm (Algorithm 1). Before that, we provide a list of all the variables and functions used by the algorithm:

- **toBeCleared**: Number of genes to be cleared;
- **activeGenes(x)**: Returns the number of genes that have its allele equals to 1;
- **noCredit(x)**: Returns the number of genes that have its allele equals to 1 and no credit;
- **clearGene(x)**: Clear an arbitrary gene that has no credit.

One could picture this scenario as if there were two forces driving the metaheuristic search towards the ideal density: one propelled by the evolutionary process on its own, rising up the density levels of its individuals, and the other exerted by the density control mechanism, pushing those levels down. We will show in our computational experiments that a proper balance between these two forces can result in better levels of performance of the GS methodology as a whole.

For the sake of clarity, a simple example is presented below. Let us consider two individuals, \(c_1\) and \(c_2\) (Tables 1 and 2). The bold and italic values indicate the activated genes of \(c_1\) and \(c_2\), respectively, representing variables. 
Algorithm 1 Density Reduction Algorithm

\[
\text{toBeCleared} \leftarrow \text{activeGenes}(x) - \text{IDD} \times \text{sizeOf}(x)
\]
\[
\text{clearedGenes} \leftarrow 0
\]

while noCredit(x) > 0 and clearedGenes < toBeCleared do
  clearGene(x)
  clearedGenes \leftarrow clearedGenes + 1
end while

effectively used in the solution generated by the SRI. The fitness value of \( c_1 \) is \( f_1 \), and that of \( c_2 \) is \( f_2 \), with \( f_1 > f_2 \).

The second row in each table (chromosome) shows the credit assigned to the genes.

### Table 1: Chromosome \( c_1 \)

<table>
<thead>
<tr>
<th>Allele</th>
<th>Credit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 1 0 1 1 0 1 1</td>
<td>( f_1 ) 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

### Table 2: Chromosome \( c_2 \)

<table>
<thead>
<tr>
<th>Allele</th>
<th>Credit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 0 0 0 0 0 0</td>
<td>( f_1 ) 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

### Table 3: Chromosome \( c_3 \)

<table>
<thead>
<tr>
<th>Allele</th>
<th>Credit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 1 0 0 0 0 0 0</td>
<td>( f_1 ) 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

### Table 4: Chromosome \( c_4 \)

<table>
<thead>
<tr>
<th>Allele</th>
<th>Credit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1 1 1 1 0 1 1</td>
<td>( f_1 ) 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

Now suppose that after crossover two offspring were generated, \( c_3 \) and \( c_4 \) (Tables 3 and 4). In \( c_3 \) and \( c_4 \), the bold values indicate the genes that came from \( c_1 \), while the italic values represent those that came from \( c_2 \). As the underlined genes have no credit, they are the natural candidates for being deactivated by DCO so as to make the individual’s density closer to the ideal density IDD. Only after all the underlined genes have been considered for deactivation is that the italic, followed by the bold, genes will be considered for deactivation, since \( f_1 > f_2 \) in this hypothetical example.

### 6. Control Parameters and Calibration

In this section we discuss the parameters that control many aspects of the GS methodology, including parameters related to the new Density Control Operator. Some of these parameters represent tradeoffs between opposite goals, while others simply allow the fine tuning of the efficacy and/or efficiency of the methodology for particular problem instances. Even though the set of control parameters discussed here permits the fine tuning of the methodology for a large range of optimization problems, we will see that several parameters were actually set to constant values during the course of our computational experiments, suggesting that certain parameter-value pairs tend to work well on a wide variety of problem instances.

In our presentation below we distinguish control parameters that are related to the GS methodology itself from those that are specific to the cutting problem presented in Section 4. For a careful discussion of the standard Genetic Algorithm (GA) parameters discussed here we refer the reader to [7].

#### 6.1. Common Control Parameters

##### 6.1.1. Elitism – EL

This is a standard GA control parameter. In our implementation this is a Boolean variable that enables or disables the standard elitism behavior. In other words, it tells whether or not the best individual is preserved from one gener-
ation to the next. It increases the evolving speed of the population by avoiding the eventual loss of good individuals during the application of recombination operators. In our experiments we enabled EL, obtaining gains in evolving speed without any side effect.

6.1.2. Population Size – PS
This is a standard GA control parameter. It controls the number of individuals in the GA population – there is no mechanism for controlling the variation of the population size in our implementation. Clearly, a larger population size will allow more diversity in the function landscape exploration, thereby increasing the probability of finding good solutions. On the other hand, a larger population size implies a slower evolution cycle. By varying this parameter in the range of 10 to 150 in steps of 20, we determined that the best results were found for the value 50. The variation of the PS value within a narrower range in the vicinity of 50 presented no further benefits.

6.1.3. Crossover Rate – CR
The Crossover Rate (CR) is also a standard GA parameter, which determines the probability \(0 \leq c_r \leq 1\) that a given individual will participate in a crossover pair, thus generating potentially new offsprings. The individuals that are not recombined by crossover generate offsprings containing their original genetic information.

6.1.4. Mutation Rate – MR
Mutation Rate is another standard GA control parameter. Each gene of an individual can have its value affected with a probability given by \(m_r \ (0 \leq m_r \leq 1)\). In the binary encoding utilized in this paper genes are affected by a mutation operation if this parameter is enabled. The mutation operator has an important role in a GA as it can potentially uncover previously unexplored regions of the function landscape. However, while smaller values of MR tend to reduce the GA exploration, larger values may cause the loss of good individuals from one generation to the next, thus reducing the algorithm’s efficacy. In our experiments values in the range of 0.02 to 0.1 led to very similar results. Values above 0.1 significantly slowed down the evolution of the GA, while values above 0.2 caused the GA to become ineffective. We utilized the value of 0.05 for all problem instances.

6.1.5. Total Time to Stop – TTS
This parameter determines the maximum amount of time allowed for the GA to execute. If TTS is reached, the execution is interrupted and the best solution found so far is returned. It is determined by the user accordingly to its time availability.

6.1.6. Maximum Number of Generations – MNG
This is a very common stopping condition parameter used in GA. When the maximum number of generations is reached the GA is interrupted and the best solution found so far is returned. If desired this parameter can be set to an undetermined value in order to be disregarded.

6.1.7. Maximum Number of Non-Evolved Generations – MNNEG
This is another standard GA stopping condition. If the best individual in the population remains the same after MNNEG generations, then the execution of the GA is interrupted. We typically adjust this parameter by assigning TTS a very large value and running several (around ten) executions of the GA – in this way, we can infer a good value of MNNEG that will reduce the execution time to a value smaller than the increased TTS, while not loosing any enhancement of the best individual(s).

6.1.8. Solver Timeout – ST
This parameter controls the maximum amount of time spent in the SRI phase. Some instances with high density can require a much longer time to be solved (and thus evaluated) than the typical instance. The exact solution of such dense reduced instances can require a prohibitive amount of time to complete. In order to avoid this situation we implemented a solver timeout (ST) mechanism. The SRI will interrupt its execution after ST seconds have elapsed. In such a case, the instance (individual) being solved is assigned a ‘0’ fitness value, meaning that its computation is considered infeasible in the tolerable time interval. Due to the implementation of certain exact solver products, the
precision of this control is low for small time intervals. Typically, a solver (such as the ILOG CPLEX [20]) will open
time windows to our timeout evaluation code based on its internal execution logic. These time windows are non-
deterministic and can vary substantially. However, the duration of such time windows are normally much smaller than
the time required by the solver to complete its execution; therefore, the overhead incurred before the solver actually
interrupts its execution is acceptable. Even though we found necessary to implement this kind of mechanism during
our experiments, it is important to remark that only in rare cases the ST parameter was effectively applied. In those
cases, the execution times were often an order of magnitude above the typical numbers. The ST parameter has to be
tuned in conjunction with the IDD parameter – we will see that IDD affects the average population density, which in
turn has a large influence on the solver execution time. The tuning of the ST parameter involves the observation of the
typical solver execution times and the choice of a value that is slightly higher than that.

6.2. Cutting and packing problem specific parameters

6.2.1. Initial Density – IND

This parameter stipulates a yardstick for the density levels of the individuals of the first generation. That is, it
is expected that the average population density reaches closely this mark, although some variation is allowed. This
is an important parameter to be set: if IND is too low, the time spent in the first generations of the GRI will be kept
small, but the average population fitness value will tend to be low; on the other hand, if we set IND to a high value,
the reverse behavior should be observed – a large running time per generation, accompanied by high average fitness
values. Therefore, the tuning of this parameter should take into account the intricacies of the problem instance at hand
in order to strike a good balance between efficiency and effectiveness. This parameter is closely related to IDD, which
is described below.

6.2.2. Ideal Density – IDD

As mentioned before, this parameter is a threshold used to trigger the density reduction algorithm (Algorithm 1),
exerting a direct influence over the time of optimization effectively spent by the GRI or SRI components. Low values
of IDD should accelerate the GRI’s evolutionary process but also restrict the optimization alternatives available to
the solver – the available cutting alternatives in the context of a cutting problem. Increasing the IDD value should
approximate the density of the reduced instances to that of the original problem, providing more “opportunities” to
the solver for generating high-quality solutions; however, this decision may incur an increase in the time elapsed from
one generation to the next in the GRI. One can think of the IND parameter as a sort of “first-shot version” of IDD,
although they do not need to share the same value. The average population density could, for instance, start at a high
level due a high IND value, but then decrease swiftly or smoothly, due to the influence of a lower IDD value. The
opposite may also happen. Since the PNZCG parameter (discussed below) affects the outcome of the application
of the density reduction algorithm, the calibration of IDD should be done in conjunction with that of PNZCG. In
order to find out a proper value for IDD, we have come up, after some experimentation with different cutting problem
instances, with the following rule of thumb: Start with a low IDD value and then increase it until the limit of nearly
10% of the solver’s timeout is reached. At this point, the solver will work closely to its operational limit but without
harming too much the performance of the GRI. We have found that this rule provides an adequate balance between
the activities of the two components of the framework.

6.2.3. Preservation of Non-Zero Credit Genes – PNZCG

As discussed earlier, the PNZCG is a Boolean parameter that enables or disables the execution of the last part of
the density reduction algorithm. When PNZCG is enabled the density control process becomes more restrictive by
allowing no individual to have a density value greater than IDD. Enabling PNZCG provides a more effective control
over the density explosion problem. On the other hand, high-fitness individuals with high-density values may lose
important genetic material if PNZCG is activated. Therefore, the decision on whether or not to enable this parameter
directly relates to the complexity of the problem under consideration. For instance, in the experiments reported in the
next section, we kept PNZCG disabled when dealing with a specific class of instances but enabled when dealing with
another class.

It is important to remark that, if PNZCG is activated and a high value of IDD is chosen, an undesirable effect may
occur: an increase in the number of solver timeouts due to an increase in the number of individuals with very high
density. So, if the adoption of PNZCG is in demand, a lower value of IDD will be a better option.
Table 5: Results for the constrained non-guillotine cutting problem instances

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FekSch1-1</td>
<td>27,718</td>
<td>27,539</td>
<td>99.35%</td>
<td>8,027.20</td>
<td>27,506.25 ± 0.258.95</td>
<td>99.24%</td>
<td>7,433.97 ± 1,288.27</td>
</tr>
<tr>
<td>FekSch1-2</td>
<td>22,502</td>
<td>22,502</td>
<td>100.00%</td>
<td>494.20</td>
<td>22,235.75 ± 332.62</td>
<td>98.82%</td>
<td>9,663.89 ± 5,618.55</td>
</tr>
<tr>
<td>FekSch1-3</td>
<td>24,019</td>
<td>24,019</td>
<td>100.00%</td>
<td>178.13</td>
<td>23,932.75 ± 119.04</td>
<td>99.64%</td>
<td>4,464.51 ± 4,369.70</td>
</tr>
<tr>
<td>FekSch1-4</td>
<td>32,893</td>
<td>32,893</td>
<td>100.00%</td>
<td>1,832.50</td>
<td>32,441.33 ± 526.36</td>
<td>98.63%</td>
<td>10,368.86 ± 4,963.37</td>
</tr>
<tr>
<td>FekSch1-5</td>
<td>27,923</td>
<td>27,923</td>
<td>96.49%</td>
<td>2,413.59</td>
<td>26,601.67 ± 687.29</td>
<td>95.27%</td>
<td>13,623.48 ± 2,039.45</td>
</tr>
<tr>
<td>Ngcutfs1-1</td>
<td>29,955</td>
<td>28,032</td>
<td>93.58%</td>
<td>3,035.99</td>
<td>27,918.40 ± 186.18</td>
<td>93.20%</td>
<td>6,792.47 ± 4,026.09</td>
</tr>
<tr>
<td>Ngcutfs1-2</td>
<td>30,000</td>
<td>28,496</td>
<td>96.21%</td>
<td>1,401.02</td>
<td>28,268.17 ± 388.68</td>
<td>94.23%</td>
<td>6,176.59 ± 2,458.72</td>
</tr>
<tr>
<td>Ngcutfs1-3</td>
<td>30,000</td>
<td>27,966</td>
<td>93.22%</td>
<td>10,506.87</td>
<td>27,932.25 ± 670.50</td>
<td>93.11%</td>
<td>10,452.96 ± 2,135.74</td>
</tr>
<tr>
<td>Ngcutfs1-4</td>
<td>30,000</td>
<td>28,494</td>
<td>94.98%</td>
<td>14,401.02</td>
<td>28,117.05 ± 403.84</td>
<td>93.73%</td>
<td>8,289.14 ± 4,750.73</td>
</tr>
<tr>
<td>Ngcutfs1-5</td>
<td>30,000</td>
<td>28,677</td>
<td>95.59%</td>
<td>3,361.34</td>
<td>28,677.00 ± 000.00</td>
<td>95.59%</td>
<td>8,616.41 ± 3,234.19</td>
</tr>
<tr>
<td>Ngcutfs3-1</td>
<td>29,943</td>
<td>28,315</td>
<td>94.54%</td>
<td>14,401.60</td>
<td>28,271.20 ± 205.46</td>
<td>94.42%</td>
<td>14,425.22 ± 24.07</td>
</tr>
<tr>
<td>Ngcutfs3-2</td>
<td>29,991</td>
<td>28,046</td>
<td>93.51%</td>
<td>14,456.59</td>
<td>27,905.00 ± 062.79</td>
<td>93.04%</td>
<td>14,427.51 ± 33.44</td>
</tr>
<tr>
<td>Ngcutfs3-3</td>
<td>30,000</td>
<td>28,877</td>
<td>96.26%</td>
<td>14,403.58</td>
<td>28,777.10 ± 061.72</td>
<td>95.92%</td>
<td>14,421.18 ± 23.71</td>
</tr>
<tr>
<td>Ngcutfs3-4</td>
<td>28,170</td>
<td>26,604</td>
<td>94.44%</td>
<td>14,402.85</td>
<td>26,506.20 ± 090.36</td>
<td>94.09%</td>
<td>14,420.40 ± 24.87</td>
</tr>
<tr>
<td>Ngcutfs3-5</td>
<td>30,000</td>
<td>27,787</td>
<td>92.62%</td>
<td>14,401.28</td>
<td>27,558.40 ± 190.66</td>
<td>91.86%</td>
<td>14,418.32 ± 17.08</td>
</tr>
</tbody>
</table>

Table 6: Comparative analysis in terms of best solution achieved

<table>
<thead>
<tr>
<th></th>
<th>Novel version</th>
<th>Original version [17]</th>
<th>Beasley’s heuristics [23]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>Best Sol.</td>
<td>Percent</td>
<td>Time (s)</td>
</tr>
<tr>
<td>FekSch1-1</td>
<td>27,718</td>
<td>27,539</td>
<td>99.35%</td>
</tr>
<tr>
<td>FekSch1-2</td>
<td>22,502</td>
<td>22,502</td>
<td>100.00%</td>
</tr>
<tr>
<td>FekSch1-3</td>
<td>24,019</td>
<td>24,019</td>
<td>100.00%</td>
</tr>
<tr>
<td>FekSch1-4</td>
<td>32,893</td>
<td>32,893</td>
<td>100.00%</td>
</tr>
<tr>
<td>FekSch1-5</td>
<td>27,923</td>
<td>27,923</td>
<td>100.00%</td>
</tr>
<tr>
<td>Average</td>
<td>27,011</td>
<td>26,975.2</td>
<td>99.87%</td>
</tr>
</tbody>
</table>

7. Computational Results

In order to evaluate the performance of the proposed version of the GS hybrid methodology, we have conducted a series of computational experiments over several benchmark problems taken from the literature, and for which the optimal solution values are already known [23]. We have decided, however, to concentrate our analysis on those problem instances that appeared to be harder to solve. We refer to these problems here as the “FekSch” (after the work of Fekete and Schepers [24]) and “Ngcutfs” classes of problem instances.

The proposed version of GS with the novel Density Control Operator was implemented in Java. We designed the code in a way that different classes of optimization problems have their specific features modeled separately from the GS algorithm itself, being subsequently integrated into a single Java project that makes use of the GS framework. For the computational tests reported here we have used as SRI the ILOG CPLEX Integer Programming Solver version 10.1 64 bits [20] running atop OpenSuse Linux 11.0 64 bits as operating system. The Java Virtual Machine adopted was the Sun HotSpot 64 bits on Server Mode, version 1.6.0-b105, whereas the machine used for the computational tests was an Intel Pentium Core 2 Quad 2.4GHz with 8 GBytes of RAM memory.

For collecting the results obtained for each problem instance, we ran the framework 10 times, after which some measures related to effectiveness (best, average, and standard deviation of objective function values) and efficiency (average time elapsed) were calculated.

After preliminary experiments, we have set the parameters introduced in Section 5 as follows. For the FekSch class of problem instances we have set \( \text{IND} = 0.05 \), \( \text{IDD} = 0.065 \), and \( \text{PNZCG} = \text{false} \), while for the Ngcutfs class
we have set \(IND = 0.04\), \(IDD = 0.05\) and \(PNZCG = \text{true}\). For all runs, the maximum limit on the execution time of the GS framework was of four hours.

The results achieved by the novel version of the GS methodology are presented in Table 5, whereas Table 6 provides a contrast with previous work. In those tables, “Optimum” and “Best Sol.” refer, respectively, to the objective function value of the known optimal solution and that of the best solution among those produced by all runs of the respective technique. Likewise, “Avg. Sol.” denotes the average solution value, whereas “Percent” measures how close the best (or average) solution value is to the known optimal one. Figure 2 displays the optimal cut configurations produced by GS for all problems of the FekSch class.

For all problems in the FekSch class, except given to the first instance, the novel version of the framework was able to find optimal solutions – a remarkable performance in terms of effectiveness. Actually, for the first instance, the best (quasi-)optimal solution achieved is very close to the true one (only 0.65% in deviation). Besides, the average solutions achieved by the novel version of the methodology are better than the best solutions found by Fekete and Schepers [24] in all cases; they are also better than those achieved by the original version of the methodology [17] in most cases.

The largest number of generations evolved by the GRI in one run of GS (considering both classes of problems) was 68, a much higher number than what could be achieved previously [16, 17]. This is another indication that the Density Control Operator was indeed instrumental to leverage the performance of the framework. It is also worth mentioning that, although we have set the limit of four hours for each run while conducting our experiments, we have noticed that, by eliminating this limit, the search process conducted by the GRI could run for days without stagnation.

In a manner as to visualize how the search and density control processes are interrelated, we refer the reader to Figure 3, which shows how the fitness and density values of the best individual and the average fitness and density values of the entire population vary along the generations for two problem instances (one for each class of problems). The curves, from top to bottom, denote, respectively: the fitness value of the best individual, the average fitness value, the density of the best individual, and the average population density. Since parameter PNZCG was kept disabled for the first problem instance (FekSch1-4), the density of some fitter individuals in this case could exceed the IDD while the majority of density values remained lower. It is important to notice that the average population density could be kept under control in both cases, oscillating around the mark of 0.2. This is opposed to what was observed in the original version of the methodology, in which the average population density usually grew rapidly, as described in [16, 17].

Since the time spent in a large factory for setting up the production line of a new product is usually measured in days or weeks, we can safely say that the average time spent by the new version of the hybrid methodology to search for a (quasi-)optimal cut configuration (measured in hours) is acceptable. Moreover, for a factory that produces a hundred thousand appliances per month, the possibility of reaching a gain of 1.44% in total cost savings may have a great economic impact. Therefore, by taking into account both facts, we argue that the new version of GS presents itself as a very competitive optimization tool that could be deployed in a real-world settings.

8. Concluding Remarks

In this paper, we have introduced the Density Control Operator (DCO) as a mechanism to leverage the performance of the GS hybrid methodology, which was recently devised to deal with hard combinatorial optimization tasks, such as those related to cutting and packing problems. After reviewing the main conceptual ingredients of the methodology, the role of this novel operator has been discussed, along with some control parameters directly related to its application. We have shown that, with the use of the DCO mechanism, it is possible to provide a better balance between the activities performed by the two complementary components of the framework – namely, a metaheuristic and an exact method – bringing about long-term stability and, therefore, increased performance.

As future work we shall apply this novel version of the hybrid framework to other classes of optimization problems, such as those involving energy consumption in wireless sensor networks [25, 26]. In principle, once a good reducible structure has been determined and subproblems can be generated, the methodology is ready to be deployed. Another promising line of investigation involves the design and implementation of parallel/distributed versions of the GS framework by means of which several GRI and SRI instances could run concurrently, each one of them configured to explore a different aspect of the optimization problem at hand.
Figure 2: Best cutting configurations produced by GS (FekSch1-1(a)–FekSch1-5(e) problem instances).
Figure 3: Fitness and density evolution – FekSch1-4(a) and Ngcutfs1-1(b).

Acknowledgment

This work has been financially supported by CNPq/Brazil via grant #473454/2008-4.

References