An Evolutionary Approach with Diversity Guarantee and Well-Informed Grouping Recombination for Graph Coloring

Daniel Cosmin Porumbel [a,b], Jin-Kao Hao [a] and Pascale Kuntz [b]

a) LERIA, Université d'Angers, 2 Bd Lavoisier, 49045 Angers, France

b) LINA, Polytech'Nantes, rue Christian Pauc, 44306 Nantes, France

January 29, 2010

Abstract

We present a diversity-oriented hybrid evolutionary approach for the graph coloring problem. This approach is based on both generally applicable strategies and specifically tailored techniques. Particular attention is paid to ensuring population diversity by carefully controlling spacing among individuals. Using a distance measure between potential solutions, the general population management strategy decides whether an offspring should be accepted in the population, which individual needs to be replaced and when mutation is applied. Furthermore, we introduce a special grouping-based multi-parent crossover operator which relies on several relevant features to identify meaningful building blocks for offspring construction. The proposed approach can be generally characterized as "well-informed", in the sense that the design of each component is based on the most pertinent information which is identified by both experimental observation and careful analysis of the given problem. The resulting algorithm proves to be highly competitive when it is applied on the whole set of the DIMACS benchmark graphs.

Keywords: Graph coloring, diversity control, population management, multi-parent crossover, memetic and hybrid algorithm.

1 Introduction

The graph coloring problem was one of the first problems proved to be NP-complete at the beginning of computational complexity studies [27]. In practical terms, graph coloring has widespread applications in areas such as timetabling, scheduling, register allocation in compilers, frequency assignment in cellular networks, and many others—see also the introduction of [30] or [2]. The first coloring algorithms date back to the 1960s [6,43] and, since then, important progress has been made. Nowadays, the literature contains a great number of heuristic algorithms that belong to three main solution approaches: sequential construction (very fast methods but not particularly efficient), local search heuristics (tabu search [2, 11, 13, 15, 24, 35], simulated annealing [3, 25], iterated local search [4, 5],

variable neighborhood search [1, 23, 41]) and population-based methods [10, 13, 15, 17, 28, 30]. A comprehensive survey of the main methods can be found in [16, 32].

From an experimental point of view, the second DIMACS Implementation challenge [26] introduced a set of graphs that become a standard benchmark for graph coloring. The best results are achieved by some highly refined local search and, more often, by hybrid evolutionary algorithms. Indeed, evolutionary search represents the most prominent and powerful approach, and so, we take it as our starting point and develop new strategies that go beyond the state of the art. To this end, we present new methods to address two fundamental issues governing the performance of evolutionary algorithms in general: population diversity management and crossover operator design.

A long-acknowledged challenge in evolutionary computing concerns the population diversity, which is particularly critical when small size populations are used. We introduce a distance measure in the search space and we show how to use it to permanently ensure a healthy *spacing* among individuals. In addition to imposing a minimum distance between any two individuals, this distance-based population management also ensures a high general dispersion (see Section 4). This strategy helps the algorithm to avoid premature convergence, while not sacrificing population quality. Additionally, our population management also reduces another important risk specific for small populations, i.e. the risk of failing to adequately cover the search space [37].

Another key aspect in evolutionary algorithms is to design a dedicated crossover that is meaningful for the given problem, i.e. that promotes good features (genes, groups) via inheritance and disrupt the bad ones [36]. For graph coloring, it is essential to view a coloring as a partition of the vertex set into a set of non-overlapping color classes [11, 15]. Indeed, most effective recombination schemes for graph coloring use color classes as building blocks and assemble different classes from parent colorings to build offspring [11, 15, 17, 22, 28, 30, 31]. In fact, this is in accordance with the principle of grouping genetic algorithms in which promising groups (or classes) are transferred from parents to offspring by inheritance [12]. An essential issue is then to correctly choose the most appropriate classes to be used for offspring construction. We introduce a refined class scoring measure determining the classes to be passed to offspring (see Section 3).

Along the paper, we also take care in determining and exploiting relevant information that can make each component of the algorithm "well-informed" and "well-founded". This concerns not only the design of the multi-parent crossover operator, but also the optimal number of parents (see Section 3.2), the optimal threshold of minimum spacing (see Section 4.3.1) imposed by the population management (see Section 4), and the moment to trigger an application of the mutation operator (see Section 4.3.2).

The resulting algorithm, hereafter called Evo–Div, proves to be highly competitive on the whole set of the DIMACS benchmark graphs. Indeed, Evo–Div is able to consistently match most of the best-known upper bounds and also to find a new solution for the largest DIMACS graph (see Section 5.2). The impact of each of the major components or techniques of the Evo–Div algorithm is also assessed (Section 5.3).

2 Hybrid Evolutionary Algorithm Design

The graph coloring problem has a very simple formulation: label the vertices of a graph with the minimum number of colors (the chromatic number) such that no two adjacent vertices share the same color. Graph k-coloring is a closely related problem: given a connected graph $G(V, E)$ and k different colors represented by numbers $\{1, 2, \ldots, k\}$, determine whether or not there is a k-coloring (a coloring using k colors) without *conflicts*, i.e. without edges with both ends of the same color. Such a conflict-free coloring is called a legal k-coloring; if there is at least one conflict, the coloring is illegal or conflicting.

A possible approach to the graph coloring (optimization) problem consists of solving it through a series of k-coloring decision problems. In this context, one starts with a sufficiently large k (e.g. $k = |V|$) and iteratively decrements k each time a legal k-coloring is found. This process stops when the algorithm can no longer find k-colorings without conflicts. Following this approach, we consider for each fixed k the associated k-coloring decision problem and we solve it as an optimization problem (see below).

2.1 Basic Algorithm Components

Encoding and search space For a given k-coloring instance defined by G and k, an individual I (chromosome) corresponds to a (legal or illegal) k-coloring which is a *partition* of V into k disjoint groups or color classes $\{I^1, I^2, \ldots, I^k\}$ such that each I^c ($c \in \{1, 2, \ldots, k\}$) contains all the vertices that are colored with color c. The search space Ω comprises all these possible k-colorings.

Notice that this group (class) oriented representation is particularly useful to avoid symmetry issues that arise if a k-coloring is encoded as an array of colors—i.e. more chromosomes can encode the same coloring. In addition, this class-based encoding constitutes the basis for designing a dedicated coloring crossover operator (see Section 3) and for defining a meaningful distance between colorings (see Section 4.2).

Fitness function Given an individual I (k-coloring), we call *conflict* (or conflicting edge) any edge having both ends in the same color class. The set of conflicts induced by I is denoted by $C_{\text{fl}}(I)$; any vertex $v \in V$, for which there exists an edge $\{v, u\}$ in $C_{\text{fl}}(I)$ is a conflicting vertex. Then the *fitness function* $f(I)$ counts the number of conflicts of I, also referred to as the *conflict number* of I. Consequently, I is a legal (conflict-free) coloring or a solution if and only if $f(I)$=0.

Crossover Our crossover operator is specially designed for the k-coloring problem and aims to inherit good features (genes) from n parents ($n \geq 2$) to the offspring. Essentially, the crossover operator selects the "best" color classes from n parents and assembles them in the offspring solution. The color classes are seen as the "building blocks" of the inheritance process; the quality of each candidate color class is assessed by a "well-informed" scoring procedure that ensures that the most meaningful information is transferred to offspring (see Section 3).

Mutation The purpose of mutation in our Evo–Div algorithm is to diversify the search. To apply a mutation, one basically moves some vertices from their initial color classes to

new classes according to a greedy procedure (see Section 4.3.2). The number of affected vertices is set according to the "needs" of our Evo–Div algorithm. While the mutation operator itself is quite straightforward, it is more important to make the evolutionary process "recognize" the most appropriate moments when mutations should be used. This issue is discussed in Section 4.3.2 about the reactive population dispersion.

Population Spacing Management Population spacing management is one of the key features of our Evo–Div algorithm. Its goal is to maintain a healthy diversity of the population all along the search process and to avoid premature convergence. As discussed in detail in Section 4, this is achieved by controlling the *spacing* with a rejection procedure (Section 4.3) and with a replacement operator (Section 4.4). A minimum spacing threshold is determined by observing the clustering of high-quality individuals (Section 4.3.1). One notices that our population spacing strategy is generally-applicable whenever a problem-specific distance is available.

2.2 Evolutionary Algorithm Template

The skeleton of Evo–Div (see Algorithm 1) shares certain ideas with previous similar hybrid or memetic algorithms [7, 10, 13, 15, 17, 28, 30, 34], but it also brings new features: strategies to control population spacing, new ways of using mutation, and well-informed crossover operator with multiple parents. In this section, we show the general Evo–Div procedure; its most important ingredients are detailed in the following sections.

Evo–Div begins with an initial population *Pop* of random individuals. For each generation, it selects randomly and uniformly n parents (`RandomParents`) whose color classes are recombined with a dedicated crossover operator (`WIPX-Crossover`) to generate an offspring O. This offspring is improved by the local search procedure. In order to ensure a minimum population diversity, the resulting individual is accepted into the population only if it fits the *spacing* criterion (`acceptOffspring`). Mutations are triggered only when the natural reproduction process can no longer produce sufficiently-spaced individuals through a number of tries (i.e. `maxRejects`). Evo–Div stops when it finds a legal coloring or reaches a predefined time limit (`Stopping-Condition`).

In addition to offspring rejection (`acceptOffspring`) and mutations (`Mutation`), Evo–Div also assures diversity by carefully choosing the individuals I_R that are eliminated from the population (`ReplacedIndiv`). Since the "survival of the fittest" selection pressure is only on the replacement operator, routine `ReplacedIndiv` becomes essential for both diversity and quality. To complete the algorithm description, the rest of the paper describes in detail the key black-box components from Algorithm 1: see Section 4 for the population management routines and Section 3 for the crossover operator. The local search procedure `LocalSearch` is independent of the evolutionary scheme and it is described below.

2.3 Local Search

The `LocalSearch` routine implements an improved version of Tabucol [24], a classical tabu search [19] algorithm for graph coloring. Numerous variants of Tabucol can be found in the literature, the reader can refer to [16] for a comprehensive survey. Our tabu

Algorithm 1 Evo–Div: Evolutionary Hybrid Algorithm with Diversity Strategy

Input: a graph $G = (V, E)$ and a positive integer k
Result: the best fitness value ever reached

1. Initialize (randomly) parent population $Pop = \{I_1, I_2, \ldots, I_{|Pop|}\}$
2. **While** *Stopping-Condition* is not met **Do**
   - **A.** $rejections = 0$
   - **B.** **Repeat**
     1. $(I_1, I_2, \ldots, I_n)$=RandomParents(Pop, n) /* $n \geq 2$ */
     2. O =WIPX-Crossover($I_1, I_2, \ldots I_n$)
     3. **If** $rejections \geq maxRejects$
        a. O =Mutation(O)
     4. O =LocalSearch(O, $maxIter$)
     5. $rejections = rejections + 1$

     Until AcceptOffspring(Pop, O)
   - **C.** I_R = ReplacedIndiv(Pop)
   - **D.** $Pop = Pop - \{I_R\} + \{O\}$

search algorithm is characterized by a refined tabu list management and a new evaluation function.

Moves and tabu tenure management Basically, the local search algorithm iteratively moves from one coloring to another by modifying the color of a conflicting vertex until either a legal coloring is found, or a predefined number of iterations (i.e. $maxIter = 100000$) is reached. Each performed move (i.e. each new color assignment) is marked tabu for a number of iterations, i.e. the tabu list length T_ℓ. Hence, Tabucol can not re-execute a move that has already been performed during the last T_ℓ iterations.

The most important particularities of our tabu search algorithm are related to the tabu list inspired by [15] and to the introduction of a new evaluation function. More precisely, $T_\ell = \alpha \cdot f(I) + \text{random}(A) + \left\lfloor \frac{L}{L_{\max}} \right\rfloor$, where α is a value in $[0, 1]$, random(A) is a function giving a random value in $\{1, 2 \ldots, A\}$, and L is the number of the last consecutive moves that kept the number of conflicts constant. The last term is a reactive component only introduced to increment T_ℓ after each series of $L_{\max}$ iterations with no fitness variation. This situation typically appears when the search process is completely blocked looping on a plateau; a longer tabu list can more easily trigger the search process diversification that is needed in this case [2]. Notice that our algorithm finally returns a random local optimum from all visited local optima having the best quality.

Well-informed evaluation function At each iteration, the tabu search algorithm has to choose a non-tabu move that leads to a coloring with the minimum number of conflicts. If there are several possible choices, the traditional approach breaks ties by making a random choice. However, we observed that it is possible to use additional information to differentiate between colorings with the same conflict number. For this purpose, we introduce the following evaluation function $\widetilde{f_{\text{eval}}}$:

$$\widetilde{f_{\text{eval}}}(I) = \sum_{\{u,v\} \in C_{\text{fl}}(I)} \left(1 - \frac{1}{2|E|\delta_u} - \frac{1}{2|E|\delta_v}\right),$$

where δ denotes the vertex degree (that is non-zero, recall G is connected) and $C_{\text{fl}}(I)$ is the set of conflicting edges (see Section 2.1). $\widetilde{f_{\text{eval}}}(I)$ can also be written as $f(I) - \frac{1}{2|E|}\sum_{\{u,v\} \in C_{\text{fl}}(I)}(\frac{1}{\delta_u} + \frac{1}{\delta_v})$, and, using $f(I) = |C_{\text{fl}}(I)| = \sum_{\{u,v\} \in C_{\text{fl}}(I)} 1$, one can check that $f(I) - 1 < \widetilde{f_{\text{eval}}}(I) \leq f(I)$. Using this function, the algorithm continues to choose moves leading to the best conflict number, but the random choice is based on this function, i.e. the lower the $\widetilde{f_{\text{eval}}}$ value of a coloring, the more chances it has to be chosen as the next coloring. Furthermore, $\widetilde{f_{\text{eval}}}$ does not introduce a significant computational overhead because it is also computed as a sum of weights of the edges in conflict—the weight of each edge $\{u, v\}$ is $\left(1 - \frac{1}{2|E|\delta_u} - \frac{1}{2|E|\delta_v}\right)$.

The basic principle behind this evaluation function is that an edge in conflict is more difficult to solve if the end vertices have higher degrees [30, 34]. If the algorithm has to choose between two 1-conflict colorings, it prefers the one with a more isolated edge in conflict, i.e. with the two conflicting vertices of lower degree, involving fewer constraints. This principle is also used in other components of the algorithm; for example, in the recombination operator we exploit such information about the degrees so as to better distinguish color classes. In Section 5.3, we will show that this principle is useful for certain graphs with a very high degree variation.

3 Well-Informed Partition Crossover (WIPX)

3.1 Rationale and Crossover Procedure

As defined in Section 2.1, a k-coloring is a partition of the vertex set of the graph. This partition interpretation of a k-coloring appears to be very useful for crossover design on this problem. A similar approach can in fact be employed for any grouping problem [12], as, for instance, bin packing and graph partitioning. In this context, the recombination consists of selecting *pertinent color classes* (groups) from the parents and then assembling them for constructing the offspring. The goal is to make the offspring inherit the best k color classes, i.e. those that bring the highest contribution on quality. For example, if one can select k conflict-free classes (independent sets) that cover V, then one can use them to construct a legal coloring. Unfortunately, such an extremely favorable situation does *not* usually occur. Consequently, an essential question needs to be addressed here: *how* to select the *best* k color classes from parents. Furthermore, instead of considering exactly two parents, we use a generalized framework that determines the appropriate number of parents for each instance.

In order to establish a *meaningful class ranking*, our first concern is to design a well-informed *scoring function for classes*. We propose to use three assessment criteria for this function: (i) the number of conflicts inside the class (`conflicts`); (ii) the number of vertices in the class (`classSize`); and (iii) the sum of the degrees of the class vertices (`degreeCls`). With these notations, the class scoring function can be formally written as $\texttt{conflicts} - \frac{1}{|V|}(\texttt{classSize} + \frac{\texttt{degreeCls}}{|E| \times |V|})$ (see Step 2.A.5 of Algorithm 2), and we

Algorithm 2 The Well Informed Partition crossover WIPX

Input: parents $I_1, I_2, \ldots, I_n$
Result: offspring O

1. O =empty, i.e. start with no vertex color assigned

2. For *currentColor*$= 1$ **To** k

A. **Foreach** parent $I_i \in \{I_1, I_2, \ldots, I_n\}$
Foreach color class I_i^c in I_i
1. Remove from I_i^c all vertices already colored in O
2. $conflicts = |\{(v_1, v_2) \in I_i^c \times I_i^c \ : \ (v_1, v_2) \in E\}|$
3. $classSize = |I_i^c|$
4. $degreeCls = \sum_{v \in I_i^c} \delta_v$ /$\star \delta_v$=degree of $v\star$/
5. $score[I_i^c] = conflicts - \frac{1}{|V|}(classSize + \frac{degreeCls}{|E| \times |V|})$

B. **Set** $(i^*, c^*) = \arg\min_{(i,c)} score[I_i^c]$

C. **Foreach** $v \in I_{i^*}^{c^*}$
$O[v] =$*currentColor*

3. Foreach unassigned $v \in O$
$O[v] = k$

argue that the best classes are associated with lower values of this function.

The first scoring criterion is essential for the offspring quality and it actually states that it is preferable to inherit classes with fewer conflicts than classes with more conflicts. Furthermore, if more classes have the same number of conflicts, they can be differentiated with the second criterion: a larger class is more valuable than a smaller one (assuming equal numbers of conflicts). Finally, there are situations where the color class sizes are quite homogeneous and a third criterion is needed for further discrimination: the sum of the degrees of the class vertices. The idea behind this third criterion is that a vertex of lower degree is more isolated and easier to color (see also Section 2.3); therefore, it is preferable to choose to inherit a colored class with higher degree vertices, leaving easier vertices uncolored.

Formally, the crossover operator, hereafter called Well-Informed Partition Crossover (WIPX), is specified in Algorithm 2. It first searches (Steps 2.A and 2.B) in all parents ($n \geq 2$) for the class with the best (minimal) score. After assigning it to the offspring (Step 2.C), it chooses the next best class and repeats. At each step, all class scores are calculated by ignoring the vertices that have already received a color in the offspring (see Step 2.A.1). WIPX stops when k colors classes are assigned; any remaining unassigned vertex receives color k (Step 3). Experiments show that the crossover operation is much less computationally expensive than the local search operator.

A potential risk of this crossover is to allow the offspring to inherit most classes only from one parent, especially if there is a (very fit) parent whose classes "eclipse" the others. However, the similarity between the offspring and the parents is implicitly checked afterwards by the diversity control procedure that rejects the offspring if it is too similar to *any* existing individual.

3.2 The Best Inheritable Features and the Number of Parents

An important principle in crossover design for grouping problems like graph coloring is to conserve "good features" through inheritance, and to disrupt the "bad" ones [12]. Our goal is then to identify what features are "good" and how to promote them in the inheritance process. Importantly, the number n of parents determines the number of classes (i.e. $n \cdot k$) from which WIPX has to select the "best" k classes (see Algorithm 2). It also determines the fragmentation of the class blending (mixture, or epistasis, see below) from each parent. By using a higher n, fewer classes would be selected from *each parent* (i.e. $\frac{k}{n}$ classes are selected from each parent—on average); and this implies a higher fragmentation/disruption of the existing class blending from each parent.

The quality of very fit colorings can reside in individual classes (genes) of high quality (e.g. independent sets), but also in their blending, or mixture. We can say that the quality can be due to excellent individual genes but also to high epistasis—e.g. productive interaction between genes. Indeed, certain instances require solutions with numerous very *small classes*—e.g. 4 vertices (on average) for an instance with $|V| = 1000$ and $k = 223$. Individual independent sets of this size do not constitute "good features": quality can only come from a productive blending of numerous small classes. This situation typically appears for dense graphs, for which one needs a large number of colors k and the average class size (i.e. $\frac{|V|}{k}$) becomes very low. For small classes, it is better to use two parents so as to avoid excessive disruption of the existing blending of color classes.

At the other extreme, there are the sparse graphs for which one needs fewer colors and *large class sizes*, i.e. 50 vertices (on average) for instances with $|V| = 1000$ and $k = 20$. In this example, the quality of very fit colorings resides rather in large independent sets (excellent genes) than in their blending. Consequently, it is preferable to use more parents because: (i) this does not disrupt good features (independent sets), and (ii) a larger number of input classes for WIPX to choose from (e.g. $n \cdot k$ with $n = 4$) increases the probability of selecting and inheriting very good individual classes (for the previous case with $k = 223$, $4 \times k \approx 1000$ input classes would have been excessive, making the result too chaotic).

In sum, our generic rule is to use two parents for instances with small classes, three for instances with average classes, or four for very large classes. To be specific, the rule employed in Evo–Div is the following one: i) $n = 2$ if $\frac{|V|}{k} < 5$, ii) $n = 4$ if $\frac{|V|}{k} > 15$, and iii) $n = 3$ otherwise.

3.3 Related Research

Algorithm 2 provides a general framework for designing recombination in coloring problems, and, more generally, in grouping or partitioning problems. In fact, by modifying the class scoring function (step 2.A.5), one can generate other crossovers. For example, to obtain a version of the Greedy Partition Crossover (GPX) from [15], one basically needs to score each class with the class size (i.e. $score = -classSize$ in step 2.A.5) and to set $n = 2$. The first three crossovers from [31] can also be replicated in this manner. The recombination from [22] uses a different framework and it is used to construct offspring only from independent sets. A similar version can be obtained by setting *score* to ∞ if the class has conflicts, or to $-classSize$ otherwise. It seems that the idea of constructing offspring only with independent sets has a positive influence on certain particular graphs;

see also Section 5.3. The approach in [17] uses a different architecture, taking classes from a central memory (and not from parents), but the class scoring function is based on the number of class vertices not yet assigned to offspring. Finally, WIPX differs from the Adaptive Multi-Parent Crossover (AMPaX) crossover of [28] in two aspects. First, like GPX of [15], AMPaX scores each class based uniquely on the class size. Second, the number of parents (ranging from 2 to 6) is determined at random with AMPaX.

In most of these methods, after inserting the k classes in the offspring, certain problematic vertices usually remain uncolored. To handle these vertices, a random or greedy procedure is commonly used to eventually assign them a color. However, this operation has a risk in disturbing good color classes because new conflicts may be introduced. We prefer to assign to all these problematic vertices the same color k (Step 3), and so, only the last color class is disturbed with new conflicts. The task of improving this assignment is thus left to the subsequent local search steps.

Note that there are also other types of crossover operators in the literature, based on color-oriented coloring encodings [13,31], on unifying pairs of conflict-free sub-classes [10]; additional ideas are available in [31].

4 Maintaining and Creating Population Diversity

4.1 Spacing Among Individuals

Our population diversity strategy has two objectives. The first one is to maintain an appropriate *minimum spacing* between individuals in order to avoid premature convergence. However, this is not enough to guarantee that the algorithm is able to *create* useful diversity, i.e. to continually discover new promising search areas. For this reason, our second objective is to make the population distribution *evolve* along time, so as to continually move from old already-visited areas to new ones. In this manner, even a small population can cover numerous areas over the time, assuring relevant diversification. Since we deal with memetic algorithms, the intensified exploitation of each new area is assured by the local search operator. There are many different means to measure diversity, but in this study we use a distance-based indicator, that is, the *spacing* S defined as the *average distance* between two individuals. In addition, the *minimum spacing* $S_{\min}$ refers to the minimum distance in the population. The two objectives of the spacing strategy can be formally expressed as follows:

- keep $S_{\min}$ above a specific minimum spacing threshold R;
- maximize the average spacing S.

Basically, the first point is addressed by the offspring rejection procedure (Section 4.3) and the second by the replacement operator (Section 4.4). They correspond to routines `AcceptOffspring` and `ReplacedIndiv` in Algorithm 1.

4.2 Search Space Distance Measure

As indicated in Section 2.1, a k-coloring is a partition of the set of vertices into k color classes. To measure the distance between two colorings (partitions), we use the well-known set-theoretic partition distance (call it d): the minimum number of elements that

need to be moved between classes of the first partition I_A so that it becomes equal to the second partition I_B. More formally, the distance $d(I_A, I_B)$ is determined using the formula $d(I_A, I_B) = |V| - s(I_A, I_B)$, where s is a similarity function defined as follows. Using the definitions from Section 2.1, $s(I_a, I_b)$ is defined by $\max_{\sigma \in \Pi} \sum_{1 \leq i \leq k} M_{i,\sigma(i)}$, where Π is the set of all permutations of $\{1, 2, \ldots, k\}$ and M is a matrix with elements $M_{ij} = |I_A^i \cap I_B^j|$ [18,21]. This similarity can be calculated by solving an assignment problem with the Hungarian algorithm of complexity $O(k^3)$ in the *worst* case. However, in our practical application, there are very few situations requiring this worst-case time complexity.

The similarity $s(I_A, I_B)$ denotes the maximum number of elements in I_A that do not need to change their class in order to transform I_A into I_B. It also reflects a structural similarity: the better the I_A classes can be associated with the I_B classes, the higher the value of $s(I_A, I_B)$ becomes; in case I_A is identical to I_B, one obtains $s(I_A, I_B) = |V|$. Both the distance and the similarity take values between 0 and $|V|$ and this is why we usually report them in terms of percentages of $|V|$. Importantly, the distance between I_A and I_B is equivalent to the minimum number of local search steps (color changes) required to go from I_A to I_B. For illustration, if $d(I_A, I_B) = \frac{|V|}{2}$, our tabu search procedure needs at least $\frac{|V|}{2}$ steps to go from one coloring to the other.

4.3 Offspring Rejection

Since the first objective of the diversity strategy is to maintain a target minimum spacing R, we insert an offspring in the population only if its distance to each existing individual in the population is greater than R. Consequently, if an offspring solution O is situated at a distance of less than R from an existing individual I, the rejection procedure detects this issue (routine `AcceptOffspring` returns `false` in Algorithm 1) and performs one of the following actions:

1. if $f(O) > f(I)$, simply reject O (O is not better than I);
2. if $f(O) \leq f(I)$, replace I with O, see also Section 4.4.1.

In both cases, since O is situated at a distance of less than R from an individual I in the population, we consider that O is not a satisfactorily diversified offspring. For this reason, Evo–Div does not pass to the next generation (Algorithm 1 does not pass to the next iteration of the `While` loop, rather it just keeps trying the crossover operation in the inner `Repeat-Until` loop) until a distanced-enough offspring solution is discovered. If it is not possible to reach such offspring after a high number of attempts of reproduction applications (i.e. `maxRejects` in Algorithm 1), a forced diversification using mutations is triggered (see Section 4.3.2).

4.3.1 How to fix the minimum spacing threshold R

A delicate issue in the above procedure is to determine a suitable spacing threshold R. Let us denote by $\mathcal{S}_R(I)$ the closed sphere of radius R centered at I, i.e. the set of individuals $I' \in \Omega$ such that $d(I, I') \leq R$. If I is a high quality individual, an appropriate value of R should imply that all other high quality individuals from $\mathcal{S}_R(I)$ share important color classes with I, i.e. they would bring no new information into the population (or they

are structurally related to I). We have to determine the "border" value of R such that all high quality individuals, that are structurally *different* from I, are situated outside $\mathcal{S}_R(I)$.

Since all individuals in the population are local minima obtained with tabu search, we can determine R by investigating the trajectory of this local search algorithm. Previous research has shown evidence that the local optima visited by tabu search are distributed in distant groups of close points (clusters) that can be confined in spheres of radius $10\%|V|$ [35]. To determine a suitable R value, it is enough to note that any two individuals situated at a distance of more than $10\%|V|$ are not in the same cluster because (ideally) they have certain different essential color classes. We presume that this observation holds on all sequences of colorings visited by tabu search and we set the value of R to $10\%|V|$. We say that two individuals distanced by less than $10\%|V|$ are "too close"; otherwise they are "R-distanced" or "distanced-enough".

4.3.2 Reactive dispersion via mutations and increased R

Applying mutations Ideally and very often, the diversity among individuals of the population can be assured only by the natural reproduction process, i.e. through crossover and local search. One of the principles of our spacing policy is to ensure diversity without sacrificing quality, and so, Evo–Div applies "artificial" mutations as rarely as possible only as a last-resort tool. The mutation itself consists of changing the color of a certain number (*mutation strength*) of randomly chosen vertices. To be specific, all colors of these vertices are first erased and new colors are sequentially assigned, using a greedy criterion that minimizes the number of generated conflicts. The mutation strength is initially fixed equal to R (i.e. R vertices are perturbed), but, if the offspring solution resulting after local search is still rejected, the strength is doubled at the next `Mutation` call (of the `Repeat-Until` loop of Algorithm 1). The mutation strength can be gradually increased (i.e. R, $2R$, $3R\ldots$) until a sufficiently high strength (at most $|V|$) enables the local search to produce a distanced offspring solution.

Recall (Algorithm 1) that the mutation is triggered only after $maxRejects$ tries that failed to produce offspring solutions distanced-enough from existing individuals. By using a very high value of $maxRejects$, Evo–Div ensures a very low overall number of mutations throughout the search (see Section 5.1.2). However, mutations can become more frequent in certain special situations where stronger diversification is necessary, i.e. when the search process is blocked looping on particular search space structures. We next show *how* to make an evolutionary search to get out of such stagnation by triggering more mutations.

Detecting stagnation and dispersion mechanism A stagnation example is given by a stable state of the population in which: (i) the average spacing in the population is less than $2R$ (see Section 4.1) and (ii) all individuals have the same fitness value (the best-ever so far). In this situation, a large part of the population can be confined in a sphere of radius $2R$ that contains many R-distanced local optima. This can be due to some particular search space structures, i.e. numerous plateaus confined in a deep "well". To overcome this difficulty, we propose a reactive dispersion mechanism that triggers numerous mutations in the subsequent generations so as to help the population to leave this problematic $2R$-sphere. This mechanism also resorts to the minimum spacing R

that is essential in the diversification/intensification balance. By doubling its value in this situation, most subsequent offspring solutions from the $2R$-sphere will be rejected. The dispersion mechanism also reduces considerably $maxRejects$ and resets $rejections$ to 0—as such, much fewer tries of natural offspring birth are allowed, resulting in more frequent mutations and more diversification.

4.4 Dispersion-Oriented Replacement Strategy

At each *generation*, an existing individual is chosen by the `ReplacedIndiv` routine in Algorithm 1 to release a slot for the offspring solution. While the offspring birth process is essential for discovering new promising areas to be explored, this replacement is also very important since it determines search areas to be abandoned.

4.4.1 Direct replacement

However, individual replacement is not carried out uniquely with the `ReplacedIndiv` routine. If the offspring solution is not distanced-enough from existing individuals, we consider that the population distribution is stagnant and Evo–Div does not pass to the next generation. As detailed in Section 4.3, if O is of better quality than I, and if the population contains an individual I "too close" to offspring O (i.e. $d(I, O) \leq R$), then I is directly replaced by O. When such direct replacement occurs, the population distribution does *not* actually evolve toward new areas, but it rather intensifies the search in the R-sphere that contains O and I. Since this R-sphere contains two high-quality individuals reached independently, we consider the R-sphere is promising enough to deserve more intensification.

This direct replacement can also result in violating the constraints of target minimum spacing, e.g. if there are I_1 and I_2 such that $d(I_1, I_2) > R$, $d(O, I_1) < R$, and $d(O, I_2) < R$, directly replacing I_1 with O would lead to a minimum spacing of $d(O, I_2) < R$. However, this is an anomaly that is solved at the next call of `ReplacedIndiv`. Indeed, this routine first finds the closest individuals I_a and I_b, and, if $d(I_a, I_b) < R$, the less fit of them is eliminated. In this case, out of the three initial *close* colorings O, I_1 and I_2, only one will eventually survive, assuring that a population slot is always set free for offspring individuals exploring new areas. Consequently, population stagnation is avoided.

4.4.2 Standard replacement

In standard cases where $d(I_a, I_b) > R$, $\forall I_a, I_b \in Pop$, the main objective of the `ReplacedIndiv` routine is to increase the average spacing and, for this, it needs to get rid of small distances between existing individuals. In addition, it should also respect the "survival of the fittest" principle. Since the parent selection is uniformly random, the replacement stage is essential for both spacing and quality.

Generally speaking, the standard elimination procedure (see Algorithm 3 below) selects two very close individuals that are candidates for elimination and only the less fit of them is eliminated. The first candidate C_1 is chosen by a random function using certain fitness-based guidelines (via the `AcceptCandidate` function). The second candidate C_2 is chosen by introducing the following *spacing criterion*: C_2 is the closest individual to C_1 respecting the same fitness-based guidelines as C_1.

Algorithm 3 The replacement (elimination) function

```
Input:  population Pop = (I_1, I_2, ..., I_|Pop|)
Result:  the individual to be eliminated
1. Repeat
       C_1 = RandomIndividual(Pop)
   Until AcceptCandidate(C_1) (fitness-based acceptance)
2. minDist = maximum possible integer
3. Foreach I ∈ Pop − {C_1}
       If d(I, C_1) <minDist
           If AcceptCandidate(I)
             • minDist = d(I, C_1)
             • C_2 = I
4. If f(C_1) < f(C_2)
       Return C_2
   Else
       Return C_1
```

The `AcceptCandidate` function separates the first half of the population from the second half—with respect to the *median* fitness value; additionally, the best individuals are also treated separately. As such, this function always accepts a candidate C_i for elimination if C_i belongs to the second half, but it accepts C_i only with 50% probability if C_i belongs to the first half. Only the best individual is fully protected; it can never become a candidate for elimination—unless there are too many best individuals (more than half of the population) in which case *any* individual can be eliminated. As such, the role of the first half of the population is to permanently keep a sample of the best individuals ever discovered. The first half of the population stays quite stable compared to the second half that is a diversity-oriented sub-population, changing very rapidly. This might recall the principles of scatter search, a population-based heuristic using an intensification set and a diversification set—see [22] for a graph coloring application.

4.5 Related Research and Ideas

There are many methods to deal with diversity in the literature. However, the idea of explicit population spacing control using distance measures, seems (to a certain extent) overlooked in hybrid evolutionary algorithms for combinatorial optimization. Compared to other studies, a novelty of our approach is that we do *not* "sacrifice quality for diversity", and that we insist on ensuring diversity via a "natural" reproduction process, with as few mutations as possible. Furthermore, we do not only conserve existing diversity, but we insist on creating new useful diversity.

In memetic algorithms, it is a common strategy to (try to) construct offspring "different enough" from its parents. Since local search is used, there is indeed a high risk that the recombination of two very fit and close parents leads to similar solutions. A policy to deal with an offspring solution "not different enough" consists of directly applying a mutation on it [39]. Another good idea is to always mate *distant* parents, or to take care to generate the offspring solution at equal distances from each parent. For illustration, col-

oring crossovers in [15,40] impose inheriting a similar number of genes/features from each parent; other distance-preserving crossovers are quite common for many problems [14].

In diversity-guided or diversity-controlling genetic algorithms one uses an indicator of overall population diversity to choose the genetic operators and their application probability [42]. In this research thread, one does not really need a distance measure between individuals, but only a general diversity indicator. In [42], a measure of statistical dispersion is used for this purpose.

Finally, one finds several interesting ideas and connections to other evolutionary computing areas. In the context of multi-objective optimization, the crowding distance (introduced in [9]) is often used for solution ranking. However, the resulting diversity concepts are quite different as the crowding distance is measured in the objective function space. In multi-modal continuous optimization, distances are often used in the context of fitness sharing and crowding selection [8, 20, 29, 33, 38]. Evo–Div does not use fitness sharing, as it does not change fitness values according to distances. A difference between our approach and the crowding selection schemes is that our replacement operator is completely separated from the offspring acceptance phase, i.e. the offspring is not taken into consideration when selecting the replaced individual. Unlike in crowding, Evo–Div does not aim at forcing offspring solutions to "replace individuals that are similar genomically" [38]

5 Experiments and Results

5.1 Experimental Conditions

5.1.1 The DIMACS benchmark

The complete DIMACS competition benchmark [26] comprises 46 graphs from the following families: (i) random graphs $dsjc$X.Y with X vertices and density Y; (ii) two families of random geometrical graphs generated by picking points uniformly at random in the unit square and by joining any 2 points distanced by less than a threshold—$dsjr$X.Y and rX.Y, where $X = |V|$ and Y is the threshold (an additional suffix "c" denotes the complementary graph); (iii) Leighton graphs leX.Y with X=450 vertices and with known chromatic number Y (they have a clique of size Y); (iv) flat graphs $flat$X.Y generated by joining vertices only between k_p predefined classes of vertices (X= $|V|$ and Y is the chromatic number k_p); (v) class scheduling graphs ($school1$, $school1.nsh$) and a latin square graph ($latin\_square$); (vi) very large random graphs ($C2000.5$ and $C4000.5$) with up to 4 million edges.

For each of the 28 DIMACS graphs in Table 1, the indicated number of colors k^* represents the best upper bound that we know of. For these graphs, Evo–Div reaches a legal coloring of value k^* in a time of seconds or minutes. Numerous other algorithms can find legal colorings for the same value of k^*. The optimality of k^* was never proved (except for Leighton and flat graphs), but we have no information about any solution with $k^* - 1$ colors. In what follows, we concentrate only on the remaining 18 graphs (see Tables 3–6), as most recent coloring papers do.

Notice that there exists a benchmark for comparing the performances of different computers on coloring instances.[1] Our machine reported a user time of 6.37s on $r500.5.b$.

[1]Technical details about the benchmark are available at `http://mat.gsia.cmu.edu/COLOR03/`, see

G	k^*	G	k^*	G	k^*	G	k^*
$dsjc125.1$	5	$r125.1$	5	$le450.5a$	5	$flat300.20$	20
$dsjc125.5$	17	$r125.5$	36	$le450.5b$	5	$flat300.26$	26
$dsjc125.9$	44	$r125.1c$	46	$le450.5c$	5	$flat1000.50$	50
$dsjc250.1$	8	$r250.1$	8	$le450.5d$	5	$flat1000.60$	60
$dsjc250.5$	28	$r1000.1$	20	$le450.15a$	15	$school1$	14
$dsjc250.9$	72	$dsjr500.1$	12	$le450.15b$	15	$school1.nsh$	14
				$le450.15c$	15		
				$le450.15d$	15		
				$le450.25a$	25		
				$le450.25b$	25		

Table 1: Easy DIMACS k-coloring instances (i.e. pairs (G, k^*)). Notice that k^* is the best known upper bound for G, but not necessarily the optimum.

5.1.2 Parameters

Parameter setting is not particularly difficult for Evo–Div; each parameter can be assigned an appropriate value only by following explicit theoretical guidelines. By searching a perfect optimal value for each parameter, one could skew the results slightly more in Evo–Div's favor, but not enough to upset our main conclusions. Table 2 summarizes the main parameters and we recall below certain design considerations that provide all needed values.

General genetic parameters: (i) population size is $|POP|$=20 (like most reported hybrid evolutionary algorithms), (ii) the number of parents n is self-tuned between 2 and 4 (Section 3.2), and (iii) minimum spacing threshold is $R = 10\%|V|$ (Section 4.3.1).

Parameters	Section	Role	Values
$\|Pop\|$	Algo. 1	Population Size	20
n	§3.2	Nr. of parents (automatically tuned)	2–4
R	§4.3.1	Target minimum spacing	$10\%\|V\|$
$maxIter$	Algo. 1	Nr. of local search iterations	100.000
$maxRejects$	§4.3.2	Nr. of rejections before using mutations	50

Table 2: Summary of evolutionary parameters.

Local search parameters The number of iterations is set to $maxIter = 100000$ and the internal parameters have been set to: $A = 10$, $\alpha = 0.6$, and $L_{\max} = 1000$, following previous work on tabu search (see also Section 2.3).

Special case parameters: (i) $maxRejects$—the maximum number of rejected offspring before resorting to mutations, and (ii) mutation strength. By using $maxRejects = 50$, we are sure to respect the principle of preserving diversity without sacrificing quality because quality-deteriorating mutations are performed very rarely (i.e. in less than 0.1% of generations for any instance). Notice that the average of "rejections per generation" is less than 1.25 for all but three graphs, as indicated by $\frac{\text{crossovers–generations}}{\text{generations}}$ in Table 3. Only during the dispersion phase, we divide $maxRejects$ by 10 to allow mutations more easily. The mutation strength is set so as to perturb R vertices (e.g. 10% of $|V|$), enough to produce a mutated coloring outside the R-sphere of the initial coloring. Another guideline

also [32].

is to use a gradually-increasing strength—i.e. if the first offspring produced via mutation is rejected, the next one perturbs $2R$, and then $3R$, $4R$, etc (see also Section 4.3.2).

5.2 Computational Results

5.2.1 Standard results with a time limit of 300 minutes

Table 3 presents the standard results of Evo–Div on 18 difficult instances with a time limit of 300 minutes (5 hours). Columns 1 and 2 denote the instance, i.e. the graph and the number of colors k (we show results only for the lowest k for which Evo–Div finds at least a solution). For each instance, this table reports the success rate over 10 independent runs (Column 3), the average number of generations required to find a solution (Column 4), the average number of crossovers (Column 5) and the average CPU time in minutes (last column). All averages take into consideration only successful runs. The total number of local search iterations is in close relation with the number of crossovers because the local search procedure (with $maxIter = 100000$) is applied once after each crossover.

Graph (k^*)	k	Successes/Runs	Generations	Crossovers	Time[m]
$dsjc500.1$ (12)	12	10/10	301	428	1
$dsjc500.5$ (48)	48	10/10	370	373	7
$dsjc500.9$ (126)	126	8/10	1987	2157	63
$dsjc1000.1$ (20)	20	10/10	1658	2454	29
$dsjc1000.5$ (83)	83	9/10	2148	2439	136
$dsjc1000.9$ (223)	223	2/10	2872	3296	245
$dsjr500.1c$ (85)	85	9/10	562	4156	93
$dsjr500.5$ (122)	122	8/10	1028	2230	36
$r250.5$ (65)	65	9/10	3175	6423	48
$r1000.1c$ (98)	98	10/10	593	2240	98
$r1000.5$ (234)	238	9/10	953	1785	99
$le450.25c$ (25)	25	10/10	10029	14648	90
$le450.25d$ (25)	25	10/10	5316	7115	45
$flat300.28.0$ (28)	31	10/10	46	50	0
$flat1000.76.0$ (82)	82	10/10	1646	1884	110
$latin\_square$ (98)	100	1/10	585	973	42
$C2000.5$ (150[148][a])	148	4/10	5051	8953	2148
$C4000.5$ (280[272][b])	271	1/10	5960	29709	32142

Table 3: Detailed results of Evo–Div with a CPU time limit of 300 minutes on the set of 18 DIMACS hard instances. The algorithm reaches most of the best known results (k^*) with a very high success rate (see Column 3). The minimal value of k for which a solution has ever been reported (i.e. k^*) is given in Column 1, in parenthesis.

[a] For this large graph, we used a time limit of 3 days; however, even 24 hours were enough for Evo-Div to find a first solution. Notice that the 148-coloring was only reported very recently and independently in [28] within a time limit of 5 days.

[b] For this exceptional large graph and for this very low k, we used a time limit of 30 days; however, legal 272-colorings can be reached in less than 10 days. Notice that the 272-coloring was only reported very recently in [28] within a time limit of 5 days.

Even if a time limit of 5 hours is not very high for graph coloring, Evo–Div finds

most of the best-known solutions from the literature, see also a comparison with other algorithms in Table 6. If we consider the complete DIMACS benchmark from Section 5.1.1, Evo–Div matches the previously best-known results for 42 graphs out of 46 and finds an improved coloring for one graph ($C4000.5$); it is below the best-known level only for three problematic instances. However, for one of these three instances ($latin\_square$), Evo–Div can actually reach the best known upper bound by intensifying the search with a longer TS chain—see Section 5.3.

We chose a time limit as the stopping condition because, in our case, machine-independent indicators are less meaningful and they can be easily misinterpreted. For example, a fixed limit on the number of generations would not take into account the computational overhead introduced by a varying number of offspring rejections and distance calculations. Furthermore, the theoretical complexity of an iteration, generation, or crossover can be different from algorithm to algorithm. The comparison of such indicators could also be biased, and so, most recent algorithms [2, 23, 30, 35] also use time stopping conditions. Our reported CPU times are obtained on a 2.8GHz Xeon processor using the C++ programming language compiled with the -O2 optimization option (gcc version 4.1.2 under Linux).

5.2.2 Results with shorter and longer time limits

We now show results with other time limits so as to better evaluate the algorithm. Table 4 presents the results using a time limit of 30 minutes (Columns 2-6) and 12 hours, respectively (Columns 7-11). The columns from this table have the same meaning as in the previous table: the interpretation of Columns 2-6 (and Columns 7-11, respectively) is the same as for Columns 2-6 in Table 3.

Some interesting conclusions can be drawn from Table 4. Evo–Div can still find many best-known solutions within only 30 minutes, which seems remarkable. Indeed, to the best of our knowledge, the smallest previous time limit giving globally competitive results is one hour (see the results in Tables 1-4 in [2] and Tables 1-3 in [23], two articles published in 2008). Evo–Div finds in 30 minutes many solutions not reached by these two algorithms from any of these tables: ($dsjc1000.5$, $k = 85$), ($flat1000.76$, $k = 83$), ($le450.25c$, $k = 25$) and ($le450.25d$, $k = 25$). This demonstrates the effectiveness of the crossover operator, as the diversity policy is less active in a short run, e.g. the percentage of rejected offspring solutions (i.e. $\frac{\#\text{cross}-\#\text{gen}}{\#\text{gen}}$) is often very low.

The results on the long run (12 hours in Table 4) show both clear improvements of the success rates and better colorings (on five very hard graphs). These results offer thus evidence that Evo–Div is capable of assuring *extensive search space coverage*. Actually, a search algorithm can get blocked on plateaus or loop between certain areas such that new areas of the search space can not be explored. In these situations, the search result will not be improved by pushing the time limit beyond a certain threshold. Thanks to the spacing policy, our Evo–Div algorithm guarantees a better exploration of the search space and thus improves its performance when more time is allowed.

Among the three instances for which Evo–Div fails to reach the best-known upper bound, one notices that only very few existing algorithms reach these bounds, only with highly specialized strategies. ($flat300.28, 28$) is solved in [35] with an intensification-oriented tabu search algorithm and in [2] with a different encoding technique, ($R1000.5, 234$)

Graph (k^*)	Time limit: 30 minutes					Time limit: 12 hours				
	k	#hits	#gen	#cross	T[m]	k	#hits	#gen	#cross	T[m]
$dsjc$500.1 (12)	12	10/10	301	428	1	12	10/10	301	428	1
$dsjc$500.5 (48)	48	10/10	370	373	7	48	10/10	370	373	7
$dsjc$500.9 (126)	126	2/10	453	514	15	126	10/10	3741	4319	125
$dsjc$1000.1 (20)	20	7/10	1466	1667	23	20	10/10	1658	2454	29
$dsjc$1000.5 (83)	85	7/10	368	368	25	83	10/10	2943	3577	178
$dsjc$1000.9 (223)	225	1/10	298	304	28	223	3/10	4559	5252	400
$dsjr$500.1c (85)	85	1/10	107	739	16	85	10/10	792	5936	136
$dsjr$500.5 (122)	122	5/10	422	593	10	122	9/10[a]	1659	4087	68
r250.5 (65)	65	6/10	650	1411	10	65	10/10	3961	10124	78
r1000.1c (98)	98	4/10	149	311	13	98	10/10	593	2240	97
r1000.5 (234)	239	5/10	326	376	24	238	10/10	1661	2639	146
le450.25c (25)	25	3/10	1660	1991	13	25	10/10	10029	14648	90
le450.25d (25)	25	4/10	1593	1926	13	25	10/10	5316	7115	45
$flat$1000.76.0 (82)	83	1/10	401	402	29	82	10/10	1646	1884	110
$flat$300.28.0 (28)	31	10/10	46	50	0	31	10/10	46	50	0
$latin\_square$ (98)	102	1/10	342	545	24	100	3/10	4189	6717	315

Table 4: Results of Evo–Div with two different time limits. In 30 minutes, Evo–Div still finds solutions not reached by other algorithms in hours. On the long run (12 hours), the diversity strategy assures a 100% success rates for most instances.

[a]For this case, 15 hours were required to reach a 10/10 success rate.

is solved in [30] thanks to column generation, ($latin\_square$, 98) is only reported in [34] with special collaborative techniques (this latter instance can actually be solved with a different Evo–Div variant, see Section 5.3 below).

Additionally, we remark that Table 4 also offers the possibility to observe the difficulty to find k-colorings for different values of k, e.g. r1000.5 can be colored by Evo–Div with $k = 239$ colors in less than 30 minutes, but it requires several hours for $k = 238$. More generally, if Evo–Div can find a solution with k colors, Evo–Div can solve the $(k + 1)$-coloring instance several times more rapidly (on average).

5.3 Influence of Diversity, Crossover and Local Search Length

In this section, we investigate the practical relevance of the most important ideas exploited by Evo–Div and the influence of the related components on the algorithm performance. For this purpose, we use ten representative graphs, that show the highest (most sensitive) performance variations.

Table 5 shows the success rate and the solving time required by Evo–Div and five other Evo–Div versions obtained by disabling certain components, as described below. The time limit is always 300 minutes. These experiments allow us to confirm our theoretical considerations on the components corresponding to the following Evo–Div versions:

1. *No-Div* (Columns 5 and 6): Evo–Div with no diversity strategy, i.e. all offspring is accepted and the worst individual in the population is eliminated at the replacement stage. Compared to Evo–Div (Columns 3 and 4), *No-Div* performs significantly

Graph (k^*)	k	Evo–Div		*No-Div*		*UnInf-Cross*		*R-5%*		*R-20%*		No-$\widetilde{f_{\text{eval}}}$	
		#hits	T[m]	#hits	T[m]	#hits	T[m]	#hits	T[m]	#hits	T[m]	#hits	T[m]
$dsjc1000.1$	20	10/10	29	0/10	–	4/10	211	9/10	37	10/10	31	10/10	26
$dsjc1000.5$	83	9/10	136	0/10	–	6/10	246	8/10	80	10/10	132	9/10	99
$dsjc1000.9$	223	2/10	245	1/10	110	2/10	220	2/10	183	0/10	–	0/10	–
$dsjr500.1c$	85	9/10	93	0/10	–	10/10	55	0/10	–	10/10	21	3/10	76
$dsjr500.5$	122	8/10	36	1/10	4	10/10	25	1/10	4	7/10	65	2/10	108
$r1000.5$	238	9/10	99	1/10	19	1/10	250	2/10	27	6/10	76	0/10	–
$le450.25c$	25	10/10	90	0/10	–	0/10	–	2/10	107	10/10	62	9/10	89
$le450.25d$	25	10/10	45	1/10	42	0/10	–	1/10	16	10/10	86	10/10	87
$flat1000.76.0$	82	10/10	110	2/10	90	7/10	236	8/10	99	7/10	159	7/10	84
$latin\_square$	100	1/10	42	0/10	–	0/10	–	0/10	–	0/10	–	1/10	211

Table 5: Comparison of the standard Evo–Div with five different versions obtained by excluding certain components. For each of these versions, we provide both the success rate (columns "#hits") and the average time in minutes (columns "T[m]"). This experiment confirms many theoretical considerations presented throughout the paper.

worse. Even if it is still able to reach some best-known legal colorings, the high success rates of Evo–Div are lost because diversity is no longer guaranteed.

2. *UnInf-Cross* (Columns 7 and 8): Evo–Div using a crossover version with $n = 2$ parents and "uninformed" class scoring function—based only on basic class size as in previous work [15], see also Section 3.3. Although *UnInf-Cross* is able to eventually find 70% of the solutions reached by Evo–Div, importantly, *UnInf-Cross* is much slower. Even by ignoring the failed 30% instances, *UnInf-Cross* can require ten times more time (see $dsjc1000.1$), and so, it obtains low success rates in most cases. However, it is still efficient for a difficult instance like ($dsjc1000.9$,$k = 223$).

3. *R-5%* (Columns 9 and 10): Evo–Div with a smaller target minimum spacing $R' = 5\%|V|$. The standard Evo–Div (with $R = 10\%|V|$) obtains systematically better results than this version. This confirms the analysis of Section 4.3.1 where it is recommended to keep a distance of $R = 10\%|V|$ between any two individuals.

4. *R-20%* (Columns 11 and 12): Evo–Div with a larger target minimum spacing $R'' = 20\%|V|$. On half of the instances, this version fails or obtains low success rates. Actually, $R'' > R$ induces excessive diversification in the search process and compromises useful intensification. In particular, it fails on ($dsjc1000.9$, $k = 223$), an instance requiring strong intensification—its first solution was found with an intensification-oriented tabu search algorithm [35] and it can even be solved with the *No-Div* version of Evo–Div. This confirms once again the recommendations of an optimum target minimum spacing of $R = 10\%|V|$.

5. *No-*$\widetilde{f_{\text{eval}}}$ (Columns 13 and 14): For this version, Evo–Div disables the degree part of the degree-based evaluation function $\widetilde{f_{\text{eval}}}$ in local search (see Section 2.3) and uses only the conflict number (i.e. f) as its evaluation function. As expected, the performance difference is more visible on graphs with high degree variation, especially on the geometrical graphs (rX.Y and $dsjr$X.Y) in which the maximum degree can be with an order of magnitude higher than the minimum degree. Similarly, the

degree-based discrimination of $\widetilde{f_{\text{eval}}}$ is less visible for graphs in which the degrees are more homogeneous (Leighton graphs, most of the random graphs). Note that $\widetilde{f_{\text{eval}}}$ has a positive influence on ($dsjc1000.9$,$k = 223$) as this graph has very high degrees.

Finally, it is worth mentioning another two simple variants of Evo–Div that are able to improve the results on certain graphs. We have also tested Evo–Div with a longer TS chain ($maxIter = 10.000.000$). By only changing this parameter, Evo–Div solved ($flat300.28$, $k = 30$) with 5/10 success rate within 300 minutes, and it even reached one solution for the same graph using $k = 29$ colors; more importantly, this version can solve the very difficult instance ($latin\_square$, $k = 98$) within 7.5 hours (by allowing a larger time limit of 12 hours, the success rate is 4/30). Furthermore, a version of our crossover working exclusively with independent sets leads to improved results for some particular geometrical graphs. Using this crossover, Evo–Div reduced the solving time to seconds for ($dsjr500.5$, $k = 85$), and it also managed to color the $r1000.5$ graph with $k = 237$ colors within 300 minutes.

5.4 Comparisons with Best Performing Algorithms

Table 6 compares the performances of the ten best performing algorithms from the literature (Columns 4–13) with the results obtained by Evo–Div within a time limit of 5 hours—see Column 3, reproducing Columns 2–3 from Table 3. Notice that certain algorithms (e.g. in [30]) can solve the coloring problem directly as an optimization problem, requiring no input value of k. The other algorithms actually solve a series of k-coloring decision problems, and so, they spend additional time for superior values of k.

To interpret these results, it is useful to know that some columns from Table 6 actually *summarize* the best results of *several* algorithms or several variants of the same algorithm, making it difficult to draw an exhaustive comparison. Yet, from Table 6, one observes that Evo–Div competes very favorably with these top coloring algorithms. Indeed, if one compares Evo–Div with any other *individual* column, one finds that over these 18 hard graphs, Evo–Div can obtain three or more new solutions (smaller k) and a worse result (larger k) only for at most one graph.

6 Conclusions

We have described a spacing-oriented hybrid evolutionary approach that enables the population to preserve and create useful diversity without sacrificing quality. The spacing strategy enables our Evo–Div coloring algorithm to successfully avoid premature convergence; by using reactive dispersion mechanisms, Evo–Div is guaranteed to continually discover new promising areas and to achieve a better coverage of the search space. On the other hand, Evo–Div employs a Well-Informed Partition Crossover that exploits a wealth of information for selecting the color classes used to construct offspring. Experiments show that this hybrid evolutionary approach is both very effective and robust.

The computational performances of Evo–Div have been extensively assessed on the whole set of the DIMACS benchmark graphs. Evo–Div is able to match all previous best-know results for all but two graphs. In particular, it finds several solutions—e.g.

Graph	χ/k^*	Evo–Div		Local Search Algorithms				Population Based Algorithms					
		k	(#hits)	ILS [4,5] 2002	TS–Div/Int [35] 2010	PCol [2] 2008	VSS [23] 2008	DCNS [34] 1996	HGA [13] 1996	HEA [15] 1999	AmaCol [17] 2008	MMT [30] 2008	MCol [28] 2010
		2009											
$dsjc500.1$	?/12	12	(10/10)	12	12	12	12	—	—	—	12	12	12
$dsjc500.5$	?/48	48	(10/10)	49	48	48	48	49	49	48	48	48	48
$dsjc500.9$	?/126	126	(8/10)	126	126	126	126	—	—	—	126	127	126
$dsjc1000.1$	?/20	20	(10/10)	—	20	20	20	—	—	20	20	20	20
$dsjc1000.5$	?/83	83	(9/10)	89	85	89	88	89	84	83	84	83	83
$dsjc1000.9$	?/223	223	(2/10)	—	223	225	224	226	—	224	224	225	223
$dsjr500.1c$	84/85	85	(9/10)	—	—	85	85	85	85	—	86	85	85
$dsjr500.5$	122/122	122	(8/10)	124	—	126	125	123	130	—	125	122	122
$r250.5$	65/65	65	(9/10)	—	—	66	—	65	69	—	—	65	65
$r1000.1c$	98/98	98	(10/10)	—	98	98	—	98	99	—	—	98	98
$r1000.5$	234/234	238[237[a]]	(9/10)	—	—	248	—	241	268	—	—	234	245
$le450.25c$	25/25	25	(10/10)	26	25	25	26	25	—	26	26	25	25
$le450.25d$	25/25	25	(10/10)	26	25	25	26	25	—	—	26	25	25
$flat300.28$	28/28	31[29[a]]	(10/10)	31	28	28	28	31	33	31	31	31	29
$flat1000.76$	76/82	82	(10/10)	—	85	88	86	89	84	83	84	82	82
$latin\_square$	?/98	100[98[a]]	(1/10)	99	—	—	—	98	106	—	104	101	99
$C2000.5$	?/150[148]	148[b]	(4/10)	—	—	—	—	150	153	—	—	—	148
$C4000.5$	?/280[272]	271[b]	(1/10)	—	—	—	—	—	280	—	—	—	272

Table 6: Best colorings reached by Evo–Div *within 5 hours* and the best results of the state-of-the-art algorithms. The colorings of Evo–Div are publicly available at: `www.info.univ-angers.fr/pub/porumbel/graphs/evodiv/`

[a]For indicative purposes, notice that the number of colors can be reduced for these graphs by using different variants of Evo–Div, i.e. ($r1000.5$, $k = 237$) was solved with a crossover that works only with independent sets; ($flat300.28$, $k = 29$) and ($latin\_square$, $k = 98$) could be solved using a longer TS chain—see also Section 5.3.

[b]For the large graphs $C2000.5$ and $C4000.5$, we used a time limit of 3 and 30 days, respectively.

($dsjc1000.9$, $k = 223$), ($flat1000.76$, $k = 82$), and ($dsjr500.5$, $k = 122$)—that have been found previously only by one or two algorithms from a large literature, i.e. see the best, most recent, ten algorithms in Table 6. Such a performance can be further appreciated if one takes into account the fact that a unique Evo–Div version is used, with a rather short computing time limit and with a pre-fixed setting of its parameters. By allowing more flexible conditions for larger graphs, Evo–Div finds another two solutions that have been reached only once before: ($C2000.5$, $k = 148$) and ($latin\_square$, $k = 98$). It even manages to find for the first time a coloring with 271 colors for the $C4000.5$ graph.

To conclude, it is important to note that the spacing strategy developed in this paper (Section 4) is in fact quite general and could be used in any evolutionary hybrid algorithm provided that a meaningful distance measure between individuals is defined. Also, the principle behind the class scoring function (Section 3) can be used for designing dedicated recombination operators for other grouping or partitioning problems.

Acknowledgments: This work is partially supported by the following projects from the French Government and the "Pays de la Loire" Région: "Pôle Informatique Régional" (2000-2006), MILES (2007-2009) and Radapop (2008-2011). We thank the referees for their useful suggestions and comments.

References

[1] C. Avanthay, A. Hertz, and N. Zufferey. A variable neighborhood search for graph coloring. *European Journal of Operational Research*, 151(2):379–388, 2003.

[2] I. Blöchliger and N. Zufferey. A graph coloring heuristic using partial solutions and a reactive tabu scheme. *Computers and Operations Research*, 35(3):960–975, 2008.

[3] M. Chams, A. Hertz, and D. de Werra. Some experiments with simulated annealing for coloring graphs. *European Journal of Operational Research*, 32(2):260–266, 1987.

[4] M. Chiarandini, I. Dumitrescu, and T. Stützle. Stochastic local search algorithms for the graph colouring problem. In T.F. Gonzalez, editor, *Handbook of approximation algorithms and metaheuristics*, pages 63–1–63–17. Chapman & Hall/CRC, Boca Raton, FL, USA., 2007.

[5] M. Chiarandini and T Stützle. An application of iterated local search to graph coloring. In D. S. Johnson et al., editors, *Computational Symposium on Graph Coloring and its Generalizations*, pages 112–125, 2002.

[6] N. Christofides. An algorithm for the chromatic number of a graph. *The Computer Journal*, 14(1):38–39, 1971.

[7] D. Costa, A. Hertz, and C. Dubuis. Embedding a sequential procedure within an evolutionary algorithm for coloring problems in graphs. *Journal of Heuristics*, 1(1):105–128, 1995.

[8] K.A. De Jong. *An analysis of the behavior of a class of genetic adaptive systems.* PhD thesis, University of Michigan Ann Arbor, MI, USA, 1975.

[9] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[10] R. Dorne and J.K. Hao. A new genetic local search algorithm for graph coloring. In *PPSN 98*, volume 1498 of *LNCS*, pages 745–754. Springer, 1998.

[11] R. Dorne and J.K. Hao. Tabu search for graph coloring, t-colorings and set t-colorings. In S. Voss et al., editors, *Meta-Heuristics Advances and Trends in Local Search Paradigms for Optimization*, pages 77–92. Kluwer, 1998.

[12] E. Falkenauer. *Genetic algorithms and grouping problems.* John Wiley & Sons, Inc. New York, USA, 1998.

[13] C. Fleurent and J.A. Ferland. Object-oriented implementation of heuristic search methods for graph coloring, maximum clique, and satisfiability. In *Cliques, Coloring, and Satisfiability Second DIMACS Implementation Challenge* [26], pages 619–652.

[14] B. Freisleben and P. Merz. A genetic local search algorithm for solving symmetric and asymmetric traveling salesman problems. In *Proceedings of IEEE International Conference on Evolutionary Computation*, pages 616–621, 1996.

[15] P. Galinier and J.K. Hao. Hybrid evolutionary algorithms for graph coloring. *Journal of Combinatorial Optimization*, 3(4):379–397, 1999.

[16] P. Galinier and A. Hertz. A survey of local search methods for graph coloring. *Computers and Operations Research*, 33(9):2547–2562, 2006.

[17] P. Galinier, A. Hertz, and N. Zufferey. An adaptive memory algorithm for the k-coloring problem. *Discrete Applied Mathematics*, 156(2):267–279, 2008.

[18] C.A. Glass and A. Pruegel-Bennett. A polynomially searchable exponential neighbourhood for graph colouring. *Journal of the Operational Research Society*, 56(3):324–330, 2005.

[19] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, Boston, 1997.

[20] D.E. Goldberg and J. Richardson. Genetic algorithms with sharing for multimodal function optimization. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 41–49. Lawrence Erlbaum Associates, 1987.

[21] D. Gusfield. Partition-distance a problem and class of perfect graphs arising in clustering. *Information Processing Letters*, 82(3):159–164, 2002.

[22] J. P. Hamiez and J. K. Hao. Scatter search for graph coloring. In *Artificial Evolution*, volume 2310 of *LNCS*, pages 168–179. Springer, 2001.

[23] A. Hertz, A. Plumettaz, and N. Zufferey. Variable space search for graph coloring. *Discrete Applied Mathematics*, 156(13):2551–2560, 2008.

[24] A. Hertz and D. Werra. Using tabu search techniques for graph coloring. *Computing*, 39(4):345–351, 1987.

[25] D.S. Johnson, C.R. Aragon, L.A. McGeoch, and C. Schevon. Optimization by simulated annealing an experimental evaluation; part ii, graph coloring and number partitioning. *Operations Research*, 39(3):378–406, 1991.

[26] D.S. Johnson and M. Trick. *Cliques, Coloring, and Satisfiability Second DIMACS Implementation Challenge*, volume 26 of *DIMACS series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996.

[27] R.M. Karp. Reducibility among combinatorial problems. In R.E. Miller and J.W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

[28] Z. Lü and J.K. Hao. A memetic algorithm for graph coloring. *European Journal of Operational Research*, 203(1):241–250, 2010.

[29] S.W. Mahfoud. A comparison of parallel and sequential niching methods. In *Conference on Genetic Algorithms*, pages 136–143, 1995.

[30] E. Malaguti, M. Monaci, and P. Toth. A metaheuristic approach for the vertex coloring problem. *INFORMS Journal on Computing*, 20(2):302, 2008.

[31] E. Malaguti and P. Toth. An evolutionary approach for bandwidth multicoloring problems. *European Journal of Operational Research*, 189(3):638–651, 2008.

[32] E. Malaguti and P. Toth. A survey on vertex coloring problems. *International Transactions in Operational Research*, 17(1):1–34, 2010.

[33] B. L. Miller and M.J. Shaw. Genetic algorithms with dynamic niche sharing for multimodalfunction optimization. In *Proceedings of IEEE International Conference on Evolutionary Computation*, pages 786–791, 1996.

[34] C. Morgenstern. Distributed coloration neighborhood search. In *Cliques, Coloring, and Satisfiability Second DIMACS Implementation Challenge* [26], pages 335–358.

[35] C.D. Porumbel, J.K. Hao, and P. Kuntz. A search space "cartography" for guiding graph coloring heuristics. *Computers and Operations Research*, 37(4):769–778, 2010.

[36] N.J. Radcliffe. The algebra of genetic algorithms. *Annals of Mathematics and Artificial Intelligence*, 10(4):339–384, 1994.

[37] C.R. Reeves. Genetic algorithms for the operations researcher. *INFORMS Journal on Computing*, 9(3):231–250, 1997.

[38] R.E. Smith, S. Forrest, and A.S. Perelson. Searching for diverse, cooperative populations with genetic algorithms. *Evolutionary Computation*, 1(2):127–149, 1993.

[39] K. Sörensen and M. Sevaux. MA|PM: memetic algorithms with population management. *Computers and Operations Research*, 33(5):1214–1225, 2006.

[40] K. Tagawa, K. Kanesige, K. Inoue, and H. Haneda. Distance based hybrid genetic algorithm: an application for the graph coloring problem. In *Proceedings of the 1999 Congress on Evolutionary Computation*, pages 2325–2332, 1999.

[41] M. A. Trick and H. Yildiz. A large neighborhood search heuristic for graph coloring. In *CPAIOR 2007*, volume 4510 of *LNCS*, pages 346–360. Springer, 2007.

[42] R. K Ursem. Diversity-guided evolutionary algorithms. In *PPSN VII*, volume 2439 of *LNCS*, pages 462–471. Springer, 2002.

[43] D.J.A. Welsh and M.B. Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10(1):85–86, 1967.