Noname manuscript No.
(will be inserted by the editor)

Using Local Search to Speed Up Filtering Algorithms for Some NP-Hard Constraints

Philippe Galinier · Alain Hertz · Sandrine Paroz · Gilles Pesant

the date of receipt and acceptance should be inserted later

Abstract This paper proposes to use local search inside filtering algorithms of combinatorial structures for which achieving a desired level of consistency is too computationally expensive. Local search quickly provides supports for many variable-value pairs, thus reducing the effort required to check and potentially filter the rest of them. The idea is demonstrated on the `SomeDifferent` constraint, a graph coloring substructure. An experimental evaluation confirms its significant computational gain in many cases.

Keywords constraint programming · local search · graph coloring

1 Introduction

Constraint programming relies heavily on identifying key substructures of a problem, writing down a model for it using the corresponding constraints, and solving it through powerful inference achieved by the efficient filtering algorithms behind each constraint. But sometimes these individual substructures are still too difficult to handle because we do not have any efficient filtering algorithm for them. In other words, deciding satisfiability for some substructures is NP-hard.

Besides breaking them up into smaller tractable pieces and thereby sacrificing the possibility of more global inference, a few researchers have proposed ways to preserve such substructures. For NP-hard optimization constraints that admit an approximation algorithm, [11] introduces the concept of approximated consistency, in which cost-based filtering removes values whose associated cost provably lies beyond a certain threshold. Opting not to compromise on the level of consistency achieved, [6] studies delayed filtering, where the filtering algorithm is called at regular intervals down a search path instead of systematically at every node of the search tree.

An earlier version of this paper appeared as [2].

École Polytechnique de Montréal
C.P. 6079, succ. Centre-ville
Montreal, Canada H3C 3A7
E-mail: {philippe.galinier, alain.hertz, sandrine.paroz, gilles.pesant}@polymtl.ca

To enforce domain consistency, every remaining value in a domain must have some support in the form of a witness solution. Local search has been very successful at solving difficult, large-scale combinatorial problems. Applied to our particular substructure, it may quickly find some solutions, each solution acting as a witness for the variable-value pair appearing in it. In this way, a collection of diverse solutions can offer a support for many variable-value pairs. Because local search is an incomplete method, it can produce witnesses to prevent filtering some values but offers no help in general to confirm those that should be filtered. If only a few unsupported candidates remain, a complete method can very well be affordable to decide about them.

There has been little work on local search algorithms inside constraints. [12] use local search to perform an incomplete filtering of NP-hard constraints, in the context of the social golfer problem. By heuristically finding large cliques in an associated graph using local search, they are sometimes able to detect inconsistencies or even filter some domains for specific constraints. We use local search for a different purpose: it quickly finds a support for most of the variable-value pairs so that we can spend more time on the remaining few to achieve complete filtering.

In the remainder of the paper, we first describe a graph coloring substructure in Section 2. We then propose in Section 3 a filtering algorithm that uses a local search procedure to determine supported variable-value pairs as well as accelerating techniques based on simplification procedures. In Section 4 we report some experimental results on three types of problems: data from a workforce management problem, random data, and instances in which most variable-value pairs have to be filtered out. Finally Section 5 recalls our contribution and makes some proposals for future research.

2 A Graph Coloring Substructure

The `AllDifferent` substructure is present in many combinatorial problems. Sometimes not every pair of variables is restricted to take different values. Richter et al. [10] have studied such a substructure, which they call `SomeDifferent`, and which can be described with the following graph coloring model.

Consider a set $X = \{x_1, \cdots, x_n\}$ of variables with domains $D = \{D_1, \cdots, D_n\}$, and a graph $G = (V, E)$ with vertex set $V = \{1, \cdots, n\}$ and edge set E. We denote $D(U) = \bigcup_{v \in U} D_v$ for any $U \subseteq V$, and D_v is called the *color set* of v. A D-coloring of G is a function $c : V \to D(V)$ that assigns a color $c(v) \in D_v$ to each vertex so that $c(u) \neq c(v)$ for all edges $(u, v) \in E$. The graph G is *D-colorable* if such an assignment exists. The *list coloring problem* is to determine if a given graph G with color sets D is D-colorable. It is NP-complete, even when restricted to interval graphs [1] or bipartite graphs [7].

Every edge $(u, v) \in E$ imposes distinct values for u and v. If G is a clique, then no two vertices can have the same value, which corresponds to the familiar `AllDifferent` constraint. Otherwise, only those pairs of vertices linked by an edge are required to have distinct values, which corresponds to the more general `SomeDifferent` constraint. Following Richter et al. [10], a *point coloring* is defined as a pair (v, i) with $v \in V$ and $i \in D_v$. A point coloring (v, i) is *supported* in G if there exists a D-coloring c of G where vertex v has color $c(v) = i$. If a point coloring (v, i) is *unsupported* in G, then color i can be suppressed from D_v, and we say that the point coloring (v, i) can be *filtered* out from G. Our goal is to achieve domain consistency which corresponds to finding new domains $D'_1, \cdots, D'_n$ so that $i \in D'_v$ if and only if (v, i) is a supported point

coloring in G. In other words, the aim is to remove the values $i \in D_v$ such that (v, i) is an unsupported point coloring. As mentioned in [10], this is an NP-hard problem.

Richter et al. [10] have developed a propagation algorithm for this problem which enumerates the sets $U \subset V$ such that $|D(U)| \leq |U|$. For each such set U and each pair (v, i) such that $v \in V \backslash U$ and $i \in D_v \cap D(U)$, they check if the point coloring (v, i) can be filtered out from G by determining whether the subgraph induced by U is D-colorable when value i is removed from the color sets of the neighbors of v. This can be done, for example, by using the `Dsatur` algorithm [9] (see Section 3.3). To accelerate their algorithm, the authors have implemented simplifications procedures which will be detailed in Section 3.2.

3 Description of the filtering algorithm

In this section, we describe a filtering algorithm to achieve domain consistency for a graph coloring substructure. It uses a local search procedure called `TabuSD` described in Section 3.1. `TabuSD` is used for two purposes : it may help to get a proof that a given graph G is D-colorable; it can also be used to detect as many supported point colorings as possible in G.

The filtering algorithm also uses simplification techniques which remove some edges from G and reduce the color sets. The remaining graph G' together with the reduced color sets $D'_v \subseteq D_v$ will have the property that G is D-colorable if and only if G' is D'-colorable. Hence all colors i which are in a set D_v but not in D'_v are such that (v, i) is an unsupported point coloring that can therefore be *filtered* out from G. These simplification techniques are gathered in a procedure called `Reduction` and described in Section 3.2

We have also developed an algorithm, called `TestColorability`, which determines whether a graph G is D-colorable. This procedure first uses the above mentioned `Reduction` algorithm to reduce G and D to G' and D'. Since G is D-colorable if and only if every connected component $G_1, \cdots, G_r$ of G' is D'-colorable, it remains to determine whether all G_j $(j = 1, \cdots, r)$ are D'-colorable. For this purpose, we first make a call to `TabuSD` to possibly get a quick proof; if such a proof is not obtained, then we use the `Dsatur` algorithm [9] as proposed in [10]. The `TestColorability` procedure will be described in detail in Section 3.3. Its output is the answer "NO" if G is not D-colorable, and the answer "YES" together with the reduced graph G' and the reduced color sets D' otherwise.

The general scheme of our filtering procedure is given in Algorithm 1. We first use the `TestColorability` procedure to test whether G is D-colorable. If this is not the case, then the algorithm stops since there is nothing to filter out. Otherwise, the output of `TestColorability` is a reduced graph G' with connected components $G_1, \cdots, G_r$ and with reduced colors sets D'. We use `TabuSD` to generate the largest possible list L_j of supported point colorings in each G_j $(j = 1, \cdots, r)$. We then test whether the point colorings (v, i) that do not belong to L_j can be filtered out of G_j. This is done by fixing value i on vertex v (i.e. reducing D'_v to $D''_v = \{i\}$ while defining $D''_u = D'_u$ for all $u \neq v$) and using `TestColorability` on G_j with D''. If G_j is D''-colorable then (v, i) is a supported point coloring in G_j which can be added to L_j. Otherwise, (v, i) can be filtered out and additional point colorings are possibly also filtered out using the `Reduction` procedure.

Algorithm 1: The filtering algorithm

Input: A graph $G = (V, E)$ with color sets D
Output: The answer *NOT SATISFIABLE* if G is not D-colorable; the answer *SATISFIABLE* with reduced color sets which are domain consistent if G is D-colorable

```
Apply TestColorability to determine if G is D-colorable;
if the answer is NO then
    return NOT SATISFIABLE;
else
    Let G_1, ..., G_r be the connected components of the graph G' obtained as output
    of TestColorability together with the reduced color sets D';
    foreach j = 1, ..., r do
        Apply TabuSD on G_j with color sets D' to generate the largest possible set L_j
        of point colorings in G_j;
        foreach point coloring (v, i) of G_j that does not belong to L_j do
            Apply TestColorability to determine if G_j is D''-colorable with D''_v = {i}
            and D''_u = D'_u for u ≠ v;
            if the answer is NO then
                Remove i from D'_v;
                Apply Reduction on G_j with color sets D' to possibly filter out
                additional point colorings;
    return (SATISFIABLE, D' = (D'_1, ..., D'_n));
```

3.1 A tabu search heuristic

Given a solution space S and a function f that measures the value $f(s)$ of every solution $s \in S$, Tabu Search is an algorithm whose objective is to determine a solution s^* with minimum value $f(s^*)$ over S. For this purpose, a neighborhood $N(s)$ is defined for every $s \in S$. It corresponds to the set of *neighbor solutions* that can be obtained from s by performing a *local move*. Tabu Search generates a sequence $s_0, s_1, \ldots, s_q$ of solutions such that s_0 is an initial solution and $s_i \in N(s_{i-1})$ for $i = 1, \ldots, q$. In order to avoid cycling, a tabu list is created that contains forbidden local moves. Hence, a local move m from s to $s' \in N(s)$ can only be performed if m does not belong to the tabu list. If a local move m from s to s' belongs to the tabu list, then solution s' is a *tabu neighbor* of s. For details on Tabu Search, the reader may refer to [4].

We have developed a tabu search algorithm, called `TabuSD` which can be used to try to determine if a graph G is D-colorable, or to generate the largest possible set of supported point colorings in G. It is described in Algorithm 2 and can be seen as a natural extension of the `Tabucol` algorithm [5] that solves the classical vertex coloring problem (where all color sets are equal to $\{1, \cdots, k\}$ for a fixed k). The solution space S is the set of all functions $c : V \rightarrow D(V)$ with $c(v) \in D_v$ for all $v \in V$. Hence, a solution is not necessarily a D-coloring since adjacent vertices u and v in G can have the same color. In such a situation, we say that the edge linking u to v is a *conflicting edge*. When `TabuSD` visits a D-coloring c (i.e., a solution without conflicting edges), all pairs $(v, c(v))$ are introduced in a list L which contains all point colorings for which we have a proof that they are supported.

Let $f_1(c)$ denote the number of conflicting edges in solution c, and let $f_2(c)$ denote the number of point colorings $(v, c(v))$ that belong to L. The objective function to be minimized by `TabuSD` is defined as

$$f(c) = \alpha f_1(c) + f_2(c)$$

Algorithm 2: TabuSD

Input: A graph $G = (V, E)$ with color sets D
Output: A set L of supported point colorings
Initialisation
Generate an initial solution c by choosing a color $c(v) \in D_v$ for each vertex v;
Set $\alpha \leftarrow 1$ and $L \leftarrow \emptyset$;
while *no stopping criterion is met* **do**
  Set $f_{best} \leftarrow +\infty$;
  foreach *vertex v which is the endpoint of a conflicting edge or such that $(v, c(v)) \in L$* **do**
    foreach *color $i \in D_v \backslash \{c(v)\}$* **do**
      if *(v, i) is not a tabu move* **then**
        Let c' be the neighbor of c obtained by assigning color i to v;
        if $f(c') < f_{best}$ **then**
          Set $v_{best} \leftarrow v$, $i_{best} \leftarrow i$ and $f_{best} \leftarrow f(c')$;
        if $f(c') < f(c)$ **then**
          Break out of the two foreach loops;
  Introduce $(v_{best}, c(v_{best}))$ in the tabu list;
  Modify c by assigning color i_{best} to v_{best}; **if** $f_1(c) = 0$ **then**
    Set $L \leftarrow L \cup \bigcup_{v \in V}(v, c(v))$;
  Update α;
return L

where α is a parameter which gives more or less importance to the first component of f. It is initially set equal to 1 and is then adjusted every 10 iterations, as in [3]: if the ten previous solutions were all D-colorings of G then α is divided by 2; if all these ten solutions had conflicting edges, then α is multiplied by 2; otherwise, α remains unchanged.

A neighbor solution $c' \in N(c)$ is obtained by assigning a new color $c'(v) \neq c(v)$ to exactly one vertex v so that either v is adjacent to a vertex u with color $c(u) = c(v)$, or $(v, c(v)) \in L$. When moving from solution c to c' by assigning a new color to v, the pair $(v, c(v))$ is declared *tabu* which means that it is forbidden to reassign color $c(v)$ to v for some number of iterations. We fix the duration of the tabu status of $(v, c(v))$ to $K + \lambda\sqrt{\mid N(c) \mid}$: if $(v, c(v)) \in L$, $(v, c'(v)) \notin L$, and v is not adjacent to any vertex with color $c(v)$, then K is an integer number randomly chosen in the interval $[30, 40]$ and we set $\lambda = 50$; otherwise, K is randomly chosen in $[20, 30]$ and we set $\lambda = 1$. These parameters worked reasonably well for the instances we tested in Section 4. They were not fine tuned — small changes to them do not have a significant impact on the behavior of the algorithm since what is important is their relative magnitude, thus favoring the exploration of solutions containing unmarked point colorings.

We use a first improvement strategy. More precisely, when evaluating the solutions in $N(c)$, it may happen that a non tabu neighbor c' is reached with $f(c') < f(c)$. In such a case, we stop evaluating the neighbors of c and move from c to c'. Otherwise, `TabuSD` moves from c to the best non tabu neighbor.

The following stopping criteria are considered. If `TabuSD` is used to detect the most possible supported point colorings, then the algorithm stops as soon as L contains all point colorings of G or is not modified for `maxiter` iterations. If `TabuSD` is used to try to prove that a graph G is D-colorable, then the algorithm also stops when a D-coloring is obtained (i.e. a solution c with $f_1(c) = 0$). For our experiments, we set `maxiter`=2000.

3.2 Reduction procedures

Before applying an algorithm to filter out all unsupported point colorings of a graph G with color sets D, it might be useful to use simplification techniques which possibly remove some edges from G and reduce the color sets.

An edge is *superfluous* if its endpoints have disjoint color sets. As mentioned in [10], it is obvious that the removal of superfluous edges from G does not modify the set of supported point colorings in G.

Also, when the color set of a vertex v contains a single color i, it is possible to apply forward checking on the binary disequality constraints. This means that color i can be removed (i.e., filtered out) from the color sets of the vertices u adjacent to v, again without modifying the set of supported point colorings. Since all edges incident to v become superfluous, they can be removed from G.

By applying the above transformations as often as possible, one gets a graph G' with color sets D' and such that G is D-colorable if and only if G' is D'-colorable. Let $G_1, \cdots, G_r$ be the connected components of G. As observed in [10], it is clear that G' is D'-colorable if and only if every G'_i is D'-colorable.

The procedure that transforms G and D into G' and D' will be called `Reduction`. It was used in [10] without the reduction on colors sets which are singletons. Notice that if a color set D'_v is empty then G' is not D'-colorable. In such a case, G has no supported point colorings and the `Reduction` procedure can therefore be stopped. We present an example in Figure 1. In (a), vertex H has only one possible color: 1. So this color is deleted from the domains of the adjacent vertices, i.e. F, G, I and J, and the edges (F, H), (G, H), (H, I) et (H, J) are now superfluous and can be deleted. This produces the new graph presented in (b). We can see that vertices F and G can only use color 3, so this color is deleted from the domains of their neighbors D and E and the edges (D, G) and (E, H) become superfluous and can be deleted. The domain of vertex I contains only color 2. As this color doesn't belong to the domain of the unique neighbor of I, J, we don't modify the domain of J but the edge (I, J) becomes superfluous and can then be deleted. The domains of the vertices J and K have an empty intersection, so we can delete the edge (J, K). The reduced graph is presented in (c). To sum up, in this example, the deleted point colorings are $(F, 1), (G, 1), (I, 1), (J, 1), (D, 3)$ and $(E, 3)$. Every other point coloring is supported.

3.3 A colorability test

We have developed an algorithm, called `TestColorability`, which determines whether a graph G is D-colorable. This procedure described in Algorithm 3 first uses the `Reduction` procedure of Section 3.2 to reduce G and D to G' and D'. We then determine the connected components $G_1, \cdots, G_r$ of G'.

Notice that if a color set D'_v is empty then G' is not D'-colorable, which means that G is not D-colorable. So assume that no color set D'_v is empty. Since G is D-colorable if and only if every connected component $G_1, \cdots, G_r$ of G' is D'-colorable, it remains to determine whether all G_j $(j = 1, \cdots, r)$ are D'-colorable. For each G_j, we first apply `TabuSD` to try to get a proof that G_j is D'-colorable. If the output list L of `TabuSD` is not empty, then we know that G_j is D'-colorable. Otherwise, we solve a classical vertex coloring problem which we now describe.

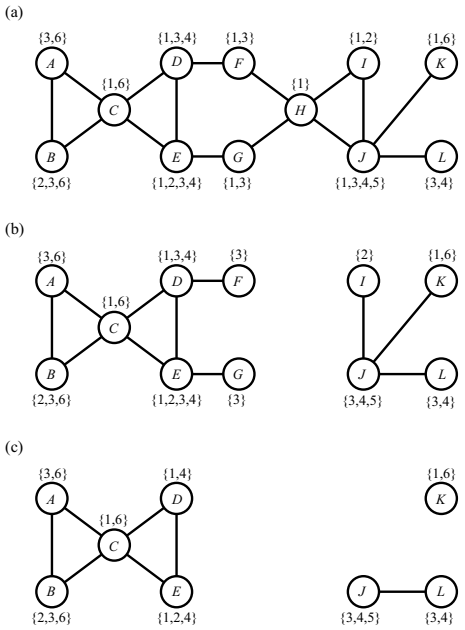

Fig. 1 Illustration of the `Reduction` procedure.

Algorithm 3: TestColorability

Input: A graph $G = (V, E)$ with color sets D
Output: The answer NO if G is not D-colorable; the answer YES together with a reduced graph G' and reduced color sets D' otherwise

Apply `Reduction` on G and D, and let G' and D' be the reduced graph and the reduced color sets obtained as output;
if *a color set* D'_v *is empty* **then**
  return *NO*
else
  Determine the connected components $G_1, \cdots, G_r$ of G';
  foreach G_j *with at least three vertices* **do**
    Apply `TabuSD` to try to determine if G_j is D'-colorable;
    if *the output list L is empty* **then**
      Construct $G \oplus D$ and use `Dsatur` to determine $\chi(G \oplus D)$;
      if $\chi(G \oplus D) > |\bigcup_{v \in G_j} D'(v)|$ **then**
        return *NO* and STOP
  return *(YES, G′, D′)*

Given a graph G with vertex set V and edge set E, the classical vertex coloring problem is to assign a color to each vertex so that no two adjacent vertices have the same color and the total number of different colors is minimized. This minimum number of required colors is called the *chromatic number* of G and is denoted $\chi(G)$. It is a special case of our D-coloring problem since for every positive integer k one can define $D_v = \{1, \cdots, k\}$, and we then have $\chi(G) \leq k$ if and only if G is D-colorable. Our purpose is to transform the D-coloring problem into a classical vertex coloring problem. This is done as in [10]. Given a graph G with vertex set V and color sets D,

we construct a new graph $G \oplus D$ from G by adding a set of $\mid D(V) \mid$ pairwise adjacent new vertices, each new vertex corresponding to a color in $D(V)$, and by linking each vertex $v \in V$ to a new vertex i if and only if $i \notin D_v$. It is then easy to observe that G is D-colorable if and only if $\chi(G \oplus D) = \mid D(V) \mid$. For determining $\chi(G \oplus D)$, we use the `Dsatur` algorithm [9] which can be downloaded from [13].

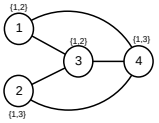

Fig. 2 A graph G before the transformation for Dsatur

Figure 2 presents a graph $G = (V, E)$ that we want to transform into $G \oplus D$. In this case $D(V) = \{1, 2, 3\}$ and so we have to add a clique with 3 vertices. These new vertices are labeled $a1$, $a2$ and $a3$, with vertex ai corresponding to color i from $D(V)$. So, vertex $a2$ will be linked with vertices 2 and 4 (because they don't have color 2 in their domain) and vertex $a3$ will be linked with vertices 1 and 3 (because they don't have color 3 in their domain). The new graph $G \oplus D$ is presented in Figure 3. The chromatic number of $G \oplus D$ is 3. Indeed, the following coloring c is legal: $c(1) = 1$, $c(2) = 1$, $c(3) = 2$, $c(4) = 3$, $c(a1) = 1$, $c(a2) = 2$ and $c(a3) = 3$. As $\chi(G \oplus D) = |D(V)|$, the graph G is D-colorable.

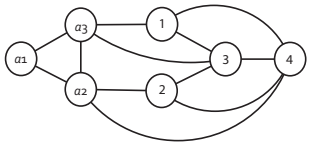

Fig. 3 The graph $G \oplus D$ where G is the graph from Figure 2.

Let G_j be a connected component of the graph produced as output of the `Reduction` procedure. If G_j contains a single vertex v, then G_j is D'-colorable since v can have any color in D'_v. If G_j contains two vertices v and u, then D'_v and D'_u are not singletons since the `Reduction` procedure isolates every vertex with a single color in its color set. Hence, for any choice of a color $i \in D'_v$ for v, there is a color not equal to i in D'_u which can be assigned to u, which means that G_j is D'-colorable. We therefore use the above transformation to a classical vertex coloring problem only if G_j contains at least three vertices.

4 Computational experiments

We evaluated our algorithm on three types of instances: real data, random graphs and graphs with a unique D-coloring. All tests were performed on a 2.80GHz Pentium D

with 1024K cache running Linux CentOS 2.6.9. Since the proposed filtering algorithm uses heuristic procedures, we performed five runs on each instance, and we report average results.

4.1 Workforce management data

The real life problem studied in [10] is a workforce management problem in a certain department at IBM. We are given a set of jobs with dates during which each job was to be performed, and a list of people qualified to perform these jobs. Jobs that overlap on times cannot be performed by the same person. This is a typical `SomeDifferent` situation which can be modeled with a D-coloring problem where the jobs are the vertices of the graph, the colors are the people, and there is an edge between two vertices if the corresponding jobs overlap in time. The color set D_v of a vertex v is the set of people qualified to perform v. In total, there are 290 instances with a number n of jobs varying from 20 to 300.

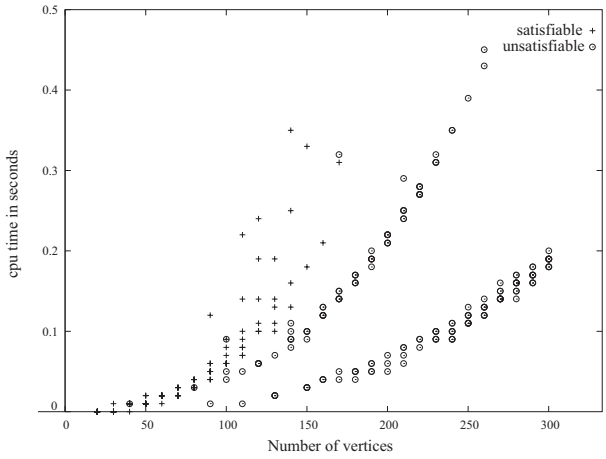

Fig. 4 Computing times for the workforce management data.

Computing times of our filtering algorithm are reported in Figure 4. The satisfiable instances are represented with a '+' sign, and the unsatisfiable ones with a '∘' sign. We observe that many instances (in particular, all those with more than 170 jobs) are not satisfiable. For the unsatisfiable instances we can identify two different curves. The lower one contains the instances for which an empty color set was generated with the `Reduction` procedure in `TestColorability`. The upper curve, which has an exponential growth, contains the instances for which a proof of unsatisfiability required a call to the `Dsatur` algorithm. While the curve for the satisfiable instances also has an exponential growth, we observe that all computing times never exceed 0.35s, similar curves being reported in [10].

We have also performed some tests on the satisfiable instances without any call to `TabuSD` to quantify the help provided by the use of a local search in the filtering process. The cpu times grew up to 10.61s and were on average 34 times slower.

In Figure 5, we give two histograms for the satisfiable instances with $n = 30, 60, 100, 170$. These instances have very few unsupported point colorings, namely 0% for $n = 30$, 0.02% for $n = 60$, 0.2% for $n = 100$, and 0.64% for $n = 170$. Supported point colorings (v, i) can be detected in three different ways, and their distribution is shown on the first histogram : if v belongs to a connected component with less than three vertices, then all colors i in D_v define a supported point coloring (v, i) (and we use label "$\in$ CC < 3"); when a color $i \in D_v$ does not appear in any color set D_u with u adjacent to v, then (v, i) is a supported point coloring (and we use label "$i \notin D_u$"); the third possibility is to obtain a proof that (v, i) is supported with `TabuSD` (and we use label "with `TabuSD`").

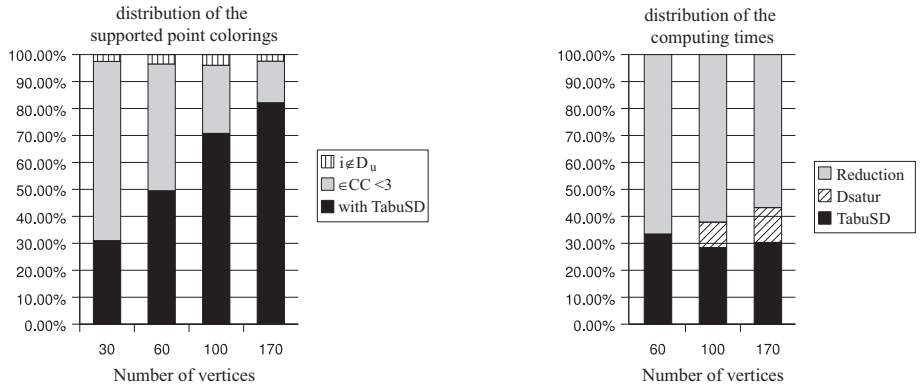

Fig. 5 Statistics on supported point colorings and computing times for some workforce management data.

We observe that most point colorings (v, i) are shown supported with `TabuSD` or with the `Reduction` procedure that creates connected components with less than three vertices. Detection with `TabuSD` increases with the size of the instance. The second histogram indicates the distribution of the computing time spent either in the `Reduction` procedure, in `TabuSD` or in `Dsatur`. We do not report the computing times for $n = 30$ since they are too small (typically less than 0.001s). We observe that `TabuSD` consumes about 30% of the total computing time. The time spent within the `Reduction` procedure decreases a little bit (from 66.7% to 56.8%) while it increases for `Dsatur` (from 0% to 12.9%) when the number of vertices increases from 60 to 170.

4.2 Random graphs

For our second test set, we considered the same random graphs as in [10]. These graphs are defined with a quadruplet (n, p, d, max_k) of parameters: n is the number of vertices, p is the probability of having an edge between two vertices, $d =\mid D(V) \mid$ is the total number of different colors, and max_k is the maximum size of a color set. For each vertex v, a integer number k_v is uniformly chosen in the interval $[1, \cdots, max_k]$, and the color set D_v for v is generated by randomly choosing k_v colors in the interval $[1, \cdots, d]$. The instances in [10] have $n = 20, 30, \cdots, 100$, $p = 0.1, 0.3, 0.6$, $d = 300$, and

$max_k = 10, 20$. We have generated additional instances considering also $n = 200, 500$, $p = 0.9$, and $max_k = 5, 40, 80$.

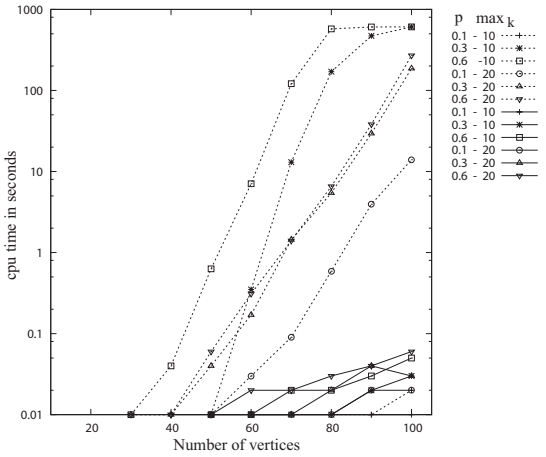

Fig. 6 Comparison of our computing times with those in [10] for random instances.

In Figure 6, we compare our computing times (represented with plain lines) to those reported in [10] (represented with dashed lines) obtained with a computer which is a little bit faster than ours. The time scale is logarithmic. We observe that our method is much faster since the maximum cpu time reported in [10] is 608.73s (for $n = 100$, $p = 0.6$, $max_k = 10$) while our filtering algorithm never requires more than 0.058s. It is not surprising that the most difficult instances are those with $p = 0.6$ since the `Reduction` procedure produces in this case a reduced graph with fewer connected components, which requires the application of `TabuSD` and `Dsatur` to larger graphs.

In Figure 7, we show the same histograms as in Figure 5, but for some random graphs. Letter "A" denotes the instances with $max_k = 10$ and $p = 0.1$, while "B" stands for $max_k = 20$ and $p = 0.3$, and "C" for $max_k = 20$ and $p = 0.6$. Again, the percentage of unsupported point colorings is very small, with a maximum of 0.6% on C instances with $n = 100$. We observe that for $max_k = 10$ and $p = 0.1$ (i.e., instances of type A), many point colorings (v, i) are shown supported because v belong to a connected component of size at most two. Moreover, for each type of parameter, A, B or C, the number of point colorings shown supported by `TabuSD` increases with the size of the instance. The second histogram does not report any statistics for $n = 40$ since the computing times are too small. Notice there is no call to `Dsatur`, this is because all supported point colorings are detected with the `Reduction` procedure or with `TabuSD`.

In Figure 8, we report all our computing times for random graphs, including those instances with $n = 200, 500$, $p = 0.9$, and $max_k = 5, 40, 80$. Again, the time scale is logarithmic. The maximum cpu time for instances having up to 100 vertices is 0.2s, which can be considered as reasonable for an algorithm used to achieve domain consistency. For random graphs with 200 vertices, the maximum cpu time grows up to 1.5s cpu, and it reaches 41.5s for $n = 500$. The increase of the cpu time is mainly due to the use of `TabuSD` which requires many iterations to find a support for almost all point colorings. For all these random graphs, we have observed that almost all point

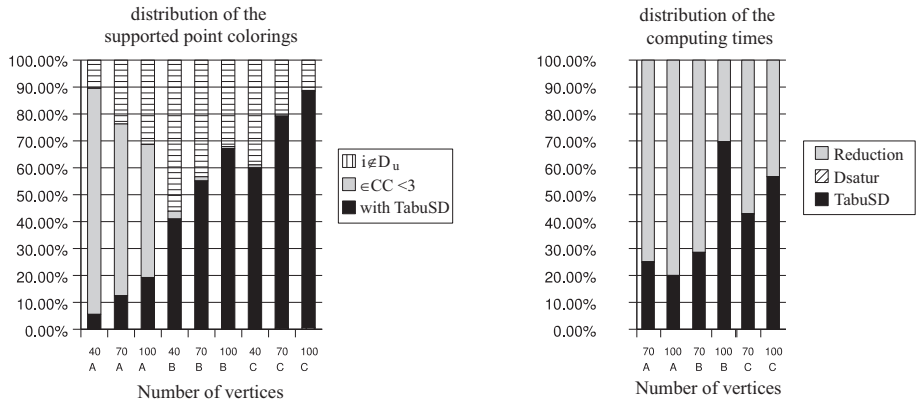

Fig. 7 Statistics on supported point colorings and computing times for random graphs. A stands for $max_k = 10, p = 0.1$, B for $max_k = 20, p = 0.3$, and C for $max_k = 20, p = 0.6$.

colorings which can be filtered out are detected during the first call to the `Reduction` procedure.

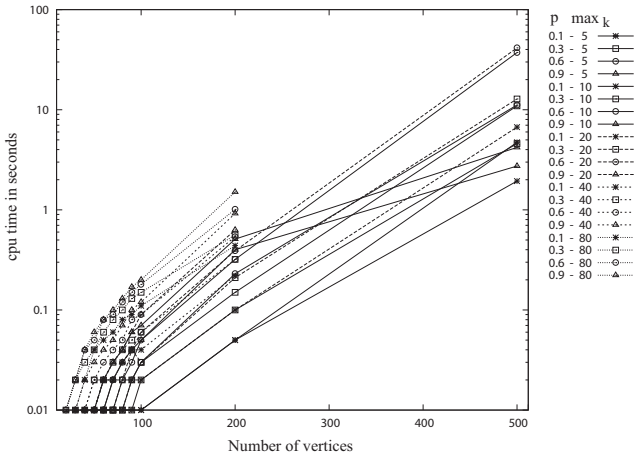

Fig. 8 Computing times of our filtering algorithm for all random graphs.

4.3 Graphs with a unique D-coloring

Mahdian and Mahmoodian [8] have described a family of graphs with a unique D-coloring. More precisely, given an integer k, they define a graph with $3k-2$ vertices, $\frac{1}{2}(9k^2-17k+8)$ edges, and in which every color set D_v contains exactly k colors. For each vertex v there is exactly one supported point coloring (v,i) with $i \in D_v$, which means that $(k-1)(3k-2) = 3k^2-5k+2$ point colorings can be filtered out. Examples of such graphs for $k=2$ and $k=3$ are represented in Figures 9 and 10. In each case,

the graph on the left is the original one and the graph on the right represents its unique vertex coloring.

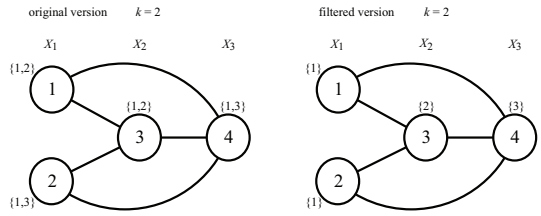

Fig. 9 Graph with a unique D-coloration as described by [8] for $k = 2$. The original graph is on the left and on the right is the filtered version.

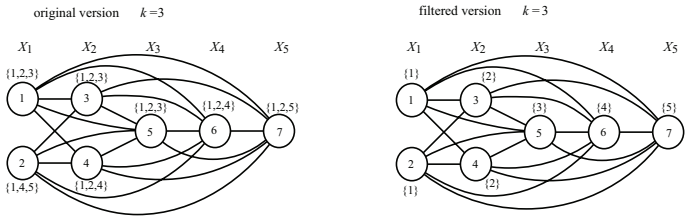

Fig. 10 Graph with a unique D-coloration as described by [8] for $k = 3$. The original graph is on the left and on the right is the filtered version.

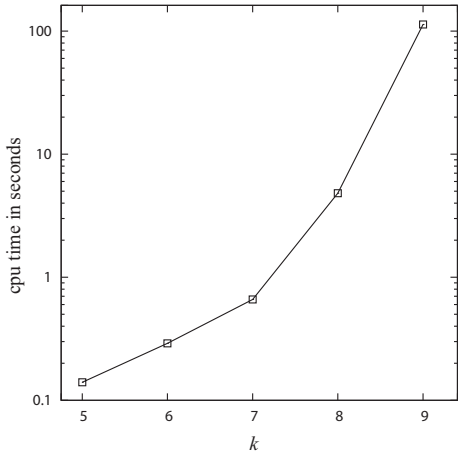

Fig. 11 Computing times on graphs with a unique D-coloring.

Computing times for $k = 5, 6, 7, 8, 9$ are reported in Figure 11. These instances are difficult ones, mainly because the `Reduction` procedure is not able to reduce the original graph and the original color sets, which means that `Dsatur` is required to

confirm that unsupported point colorings can indeed be filtered out. The exponential growth of the computing times which exceed 100 seconds for graphs with 25 vertices (for $k = 9$) shows the limit of applicability of our filtering algorithm.

The first histogram of Figure 12 indicates the percentage of point colorings which are either filtered out, or shown supported by `TabuSD` or `Dsatur`. As expected, the proportion of filtered point colorings is $\frac{k-1}{k}$. While most point colorings are shown supported with `TabuSD` for small values of k, `Dsatur` does the job for larger values since `TabuSD` is not able to find the unique D-coloring (with `maxiter`=2000). Statistics on computing times are given on the second histogram where we clearly observe that the time spent in `Dsatur` drastically increases with k.

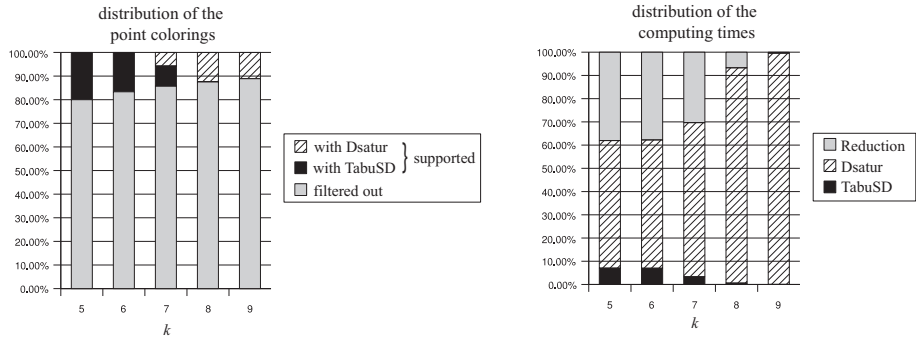

Fig. 12 Statistics on supported and unsupported point colorings and on computing times for graphs with a unique D-coloring.

5 Conclusion and future work

In this paper, we have presented a filtering algorithm for the `SomeDifferent` constraint (i.e. the list coloring problem) which combines a tabu search to quickly find a supporting solution for as many point colorings as possible, and an exact algorithm to validate or filter out the remaining point colorings. Our filtering algorithm turned out to be about as fast as the one of Richter et al. [10] when tested on data from a workforce management problem, and significantly faster for random data.

In a future work, we intend to test our implementation of the `SomeDifferent` constraint within a constraint programming model, in conjunction with other types of constraints, in order to measure the potential increased efficiency.

The general principles of the proposed approach are not specific to the graph coloring substructure. Indeed, the technique can be adapted to other NP-hard constraints in order to obtain a filtering procedure that enforces domain consistency. This can be done by developing mainly two specific (problem-dependent) low-level procedures: an exact algorithm and a local search procedure. Therefore, it will be interesting in the future to investigate this approach for other NP-hard constraints.

References

1. Miklós Biro, Mihály Hujter, and Tuza Zsolt. Precoloring extension 1. interval graphs. *Discrete Mathematics 100*, pages 267–279, 1992.
2. Philippe Galinier, Alain Hertz, Sandrine Paroz, and Gilles Pesant. Using Local Search to Speed Up Filtering Algorithms for Some NP-Hard Constraints. In Laurent Perron and Michael A. Trick, editors, *CPAIOR*, volume 5015 of *Lecture Notes in Computer Science*, pages 298–302. Springer, 2008.
3. Michel Gendreau, Alain Hertz, and Gilbert Laporte. A tabu search heuristic for the vehicle routing problem. *Management Science 40*, pages 1276–1290, 1994.
4. Fred Glover and Manuel Laguna. *Tabu Search*. Kluwer Academic Publishers, Boston, USA, 1997.
5. Alain Hertz and Dominique de Werra. Using tabu search for graph coloring. *Computing 39*, pages 345–351, 1987.
6. Irit Katriel. Expected-case analysis for delayed filtering. In *Proceedings of the Third International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR'06)*, volume 3990 of *LNCS*, pages 119–125. Springer, 2006.
7. Klaus Jansen and Petra Scheffler. Generalized coloring for tree-like graphs. *Discrete Applied Mathematics 75*, pages 135–155, 1997.
8. Mohammad Mahdian and Ebad S. Mahmoodian. A characterization of uniquely 2-list colorable graphs. *Ars Combinatoria 51*, pages 295–305, 1999.
9. Juergen Peemoeller. A correction to brélaz's modification of brown's coloring algorithm. *Communications of the ACM 26/8*, pages 593–597, 1983.
10. Yossi Richter, Ari Freund, and Yehuda Naveh. Generalizing alldifferent: The somedifferent constraint. In *Proc. 12th International Conference on Principles and Practice of Constraint Programming (CP06)*, volume 4204 of *LNCS*, pages 468–483. Springer, 2006.
11. Meinolf Sellmann. Approximated consistency for knapsack constraints. In *Proceedings of the Nineth International Conference on the Principles and Practice of Constraint Programming (CP'03)*, volume 2833 of *LNCS*, pages 679–693. Springer, 2003.
12. Meinolf Sellmann and Warwick Harvey. Heuristic constraint propagation. In *Proceedings of the Fourth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR'02)*, pages 191–204, 2002.
13. Michael Trick. `http://http://mat.gsia.cmu.edu/COLOR/solvers/trick.c`.