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

Iterated backtrack removal search for finding k-vertex-critical subgraphs

Wen Sun · Jin-Kao Hao* · Alexandre Caminada

Accepted: Journal of Heuristics, September 2017

Abstract Given an undirected graph $G = (V, E)$ and a positive integer k, a k-vertex-critical subgraph (k-VCS) of G is a subgraph H such that its chromatic number equals k (i.e., $\chi(H) = k$), and removing any vertex causes a decrease of $\chi(H)$. The k-VCS problem (k-VCSP) is to find the smallest k-vertex-critical subgraph H^* of G. This paper proposes an iterated backtrack-based removal (IBR) heuristic to find k-VCS for a given graph G. IBR extends the popular removal strategy that is intensification-oriented. The proposed extensions include two new diversification-oriented search components – a backtracking mechanism to reconsider some removed vertices and a perturbation strategy to escape local optima traps. Computational results on 80 benchmark graphs show that IBR is very competitive in terms of solution quality and run-time efficiency compared with state-of-the-art algorithms in the literature. Specifically, IBR improves the best-known solutions for 9 graphs and matches the best results for other 70 instances. We investigate the interest of the key components of the proposed algorithm.

Keywords: Vertex-critical subgraph; Graph coloring; Tabu search; Backtracking-based diversification; Irreducibly inconsistent systems.

Wen Sun
LERIA, Université d'Angers, 2 Boulevard Lavoisier, 49045 Angers, France

Jin-Kao Hao (Corresponding author)
LERIA, Université d'Angers, 2 Boulevard Lavoisier, 49045 Angers, France
Institut Universitaire de France, 1 Rue Descartes, 75231 Paris, France
E-mail: jin-kao.hao@univ-angers.fr

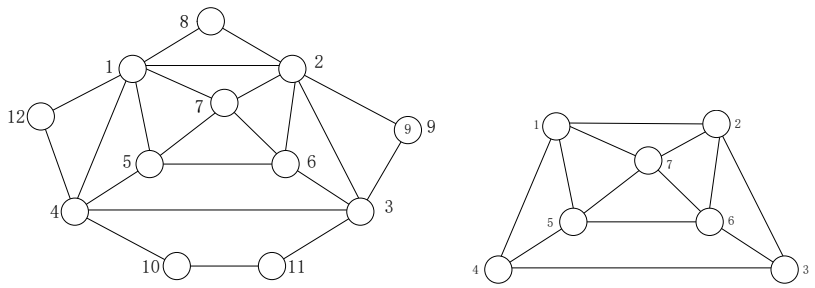
Alexandre Caminada
UTBM, Rue Thierry Mieg, 90010 Belfort, France

1 Introduction

Given a simple undirected graph $G = (V, E)$ with vertex set $V = \{1, 2, \ldots, n\}$ and edge set $E \subset V \times V$, a legal k-coloring of G is a mapping $c : V \rightarrow \{1, \ldots, k\}$, such that $c(i) \neq c(j)$ for all edges (i, j) in E. The graph k-coloring problem (K-COL) is to determine if a legal k-coloring of G exists for a given k. The classical graph coloring problem (COL) is to find the minimum integer k (chromatic number $\chi(G)$) for which a legal k-coloring of G exists. K-COL is known to be NP-complete while the optimization problem COL is NP-hard [11].

A graph is a vertex-critical graph if removing any vertex from the graph decreases its chromatic number [5]. Given an integer k, a k-vertex-critical subgraph (k-VCS) of G is a vertex-critical subgraph H such that $\chi(H) = k$. Note that each graph G contains at least one k-VCS for $1 \leq k \leq \chi(G)$. Finally, a subgraph H^* is a minimum k-VCS if no other k-vertex-critical subgraph with fewer vertices than in H^* exists in G. The k-VCS problem (k-VCSP) is to find a minimum k-vertex-critical subgraph of G. k-VCSP is a NP-hard problem and thus computationally challenging [5]. For simplicity, if $H = (A, E_A)$ is a k-VCS, we also use its vertex set A to denote the k-VCS.

As an example, consider the graph G of Figure 1(a) with $\chi(G) = 4$. Figure 1(b) shows a 4-vertex-critical subgraph of G since removing any vertex decreases its chromatic number to 3. Moreover, this 4-VCS is also minimum since no other 4-critical subgraph with fewer vertices exists in G. Note that a k-VCS of G provides a lower bound of $\chi(G)$, and a k-VCS with a larger k thus leads to a better (tighter) bound (eg., the 4-VCS of Figure 1(b) corresponds to a better lower bound with respect to any 3-VCS formed by a clique of size 3 like {1,2,7}).



(a) A graph with chromatic number of 4 (b) A 4-vertex-critical subgraph

Fig. 1 An example of k-vertex-critical subgraph.

It is worth noting that k-VCSP is tightly related to the problem of finding irreducible inconsistent sets in linear programs (IIS) [1–4,21,26], minimal unsatisfiable sets (MUSes) in infeasible propositional satisfiability problems [19, 23,24] and constraint satisfaction problems [8,15,20].

In the context of graph coloring, some effective approaches have been proposed to extract k-VCS in a graph. In [16], Herrmann and Hertz suggested a vertex removal algorithm combined with an insertion algorithm to find the chromatic number of a graph. In [5], Desrosiers et al. proposed the neighborhood weight heuristic algorithm which is combined with classical critical subgraph detection algorithms, leading to several effective k-VCS detection heuristics including the $Ins + h$ algorithm that we will use as our main reference algorithm for our computational studies.

This paper extends previous studies by proposing an Iterated Backtrack Removal Search (IBR) for finding k-VCS in a graph. IBR reinforces the general and classical removal approach [5,16] with a backtracking scheme and a perturbation strategy. The removal approach reduces the current graph by tentatively moving vertices to the set of uncritical vertices (see Section 3.2). The backtracking procedure expands the current subgraph by adding back some vertices which are incorrectly removed (see Section 3.4). The perturbation procedure provides a means of reconsidering some vertices which would have been incorrectly identified as critical ones (see Section 2). While the basic removal procedure can be considered as an intensification-oriented component, backtracking and perturbation provides two complementary means for search diversification.

We assess the proposed approach on 80 popular DIMACS and COLOR02-04 benchmark instances. We show that our IBR algorithm competes favorably with the state-of-the-art results. Specifically, our algorithm improves the best-known solutions for 9 graphs and matches the best results for other 70 instances. Only in one case, IBR obtains a slightly worse result.

The remainder of this paper is organized as follows. Section 2 introduces some useful notations. Section 3 presents the components of the IBR algorithm, including the basic removal algorithm, the backtrack-based removal algorithm and the perturbation procedure. Section 4 shows computational evaluation and comparisons with state-of-the-art results. Section 5 investigates the key components of the proposed algorithm, followed by conclusions in Section 6.

2 Notations

Let $G = (V, E)$ be an input graph, we introduce the following notations, which are useful for the description of the proposed approach.

We define three working sets of vertices A, B and C that are used by the algorithm. For each vertex of V, we first define its status (critical, unknown, uncritical).

A *critical vertex* is a vertex that belongs to a k-VCS. We use A to denote the *set of critical vertices* that have been detected. A is also called the critical set.

An *unknown vertex* is a vertex whose status is still unknown. We use B to denote the *set of unknown vertices*, which is also called the unknown set.

An *uncritical vertices* refers to a vertex that does not belong to a k-VCS. We use C to denote the *set of uncritical vertices*, which is also called the uncritical set.

Given the critical set A and the unknown set B, the subgraph induced by A, $H = G_A = (A, E_A)$ ($E_A = A \times A \cap E$), is called the *critical subgraph* of G. The subgraph induced by $A \cup B$, $G_{A \cup B} = (A \cup B, E_{A \cup B})$ ($E_{A \cup B} = ((A \cup B) \times (A \cup B)) \cap E$), is the *remaining subgraph* of G.

As explained in the next section, the proposed approach operates on these three sets of vertices, which are initially set as $A = \emptyset$, $B = V$, $C = \emptyset$. Then the algorithm searches a k-VCS by moving vertices from one set to another according to the temporarily identified status of each vertex.

For any vertex i, we also associate a weight $w(i)$, which is defined as follows:

$$w(i) = \begin{cases} |E|, if\ i \in set\ A. \\ 1, if\ i \in set\ B; \\ 0, if\ i \in set\ C; \end{cases} \quad (1)$$

At the beginning of the search, we set $w(i) = 1$ for each vertex i of V (since $A = \emptyset$, $B = V$, $C = \emptyset$). The weight of a vertex is updated each time the vertex changes its status (critical, unknown, uncritical) (see Section 3). The dynamically learned weight information is mainly used by the removal algorithm for vertex selection [5].

3 The Iterated Backtrack-based Removal Algorithm

In this section, we present the iterated backtrack-based removal algorithm (IBR) for detecting k-VCS in a graph, which extends the classical removal algorithm with a backtracking scheme and a perturbation procedure.

3.1 General Structure of the IBR Algorithm

Given a graph $G = (V, E)$ and an integer $k \leq K$ (K being $\chi(G)$ or the smallest number of colors for which a k-coloring is known to exist), the proposed IBR algorithm (see Figure 2 and Algorithm 1) aims to find a small k-VCS, i.e., a small set of critical vertices $A \subseteq V$ such that the induced subgraph $H = (A, E_A)$ has a chromatic number of $\chi(H) = k$, and removing any vertex from H decreases $\chi(H)$.

The IBR algorithm operates with three working sets of vertices, initialized as $A = \emptyset, B = V, C = \emptyset$. Basically, according to the coloring results on the remaining graph provided by a heuristic coloring algorithm, IBR uses three main procedures to move vertices among these sets in order to find a k-VCS.

Recall that each graph G contains at least one k-VCS for $1 \leq k \leq \chi(G)$. Furthermore, a graph G is $(k-1)$-colorable if and only if G does not contain

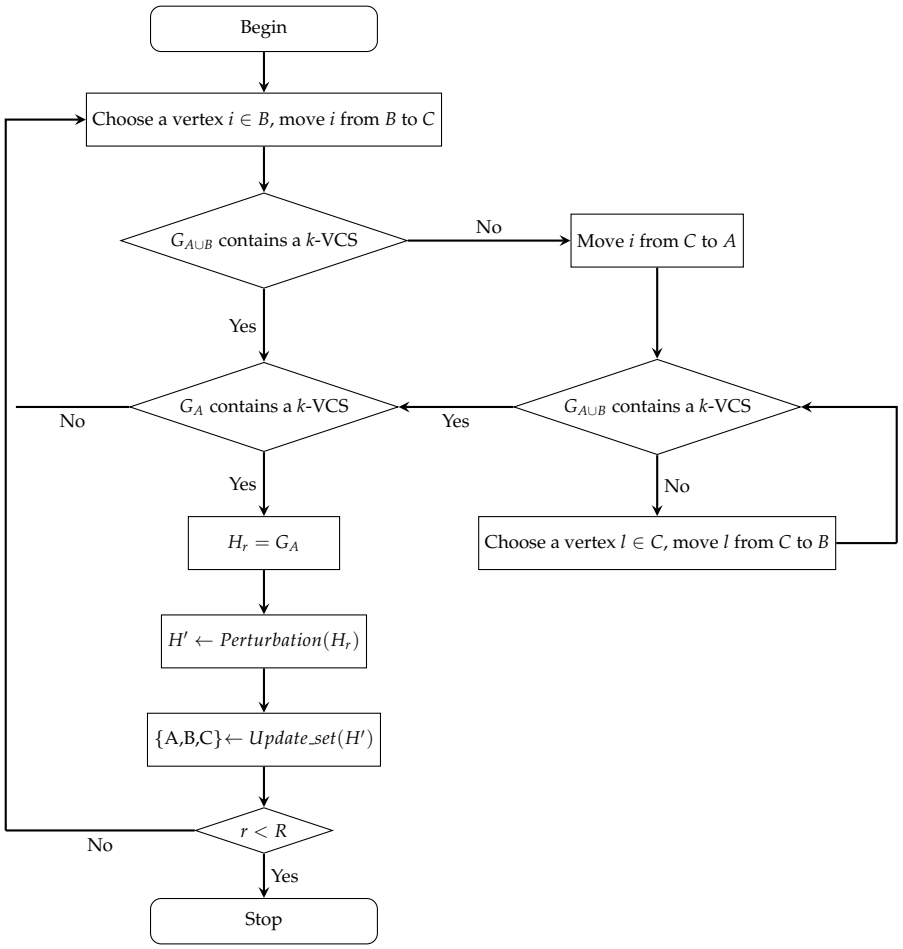


Fig. 2 Flow chart of the iterated backtrack-based removal algorithm for finding k-VCS in a graph.

a k-VCS. Note that the problem of deciding whether a graph G is $(k-1)$-colorable is itself NP-hard. For this reason, we adopt a heuristic coloring algorithm *Color* (see Section 3.3) to judge whether G is $(k-1)$-colorable.

1. The *removal procedure* inspects, one by one, the vertices of set B of unknown vertices to determine their status (Algorithm 1, lines 7-16). Specifically, at each iteration, a vertex i from B is first selected and moved to C (i.e., i is supposed to be uncritical, line 8). If the graph $G_{A\cup B}$ after removing i does not contains a k-VCS any longer, i.e., becomes $(k-1)$-colorable, (checked with a heuristic coloring algorithm, see Section 3.3), then vertex i is identified as critical and moved from C to A (lines 9-10, see Section 3.2). We check again with the heuristic coloring algorithm whether the graph $G_{A\cup B}$ contains a k-VCS (line 12). If the graph $G_{A\cup B}$ after adding i back still does not contain a k-VCS, then certain critical vertices were

Algorithm 1 The IBR algorithm for solving k-VCSP

1: **Input**: Graph $G = (V, E)$, integer $k \leq K$ (K being $\chi(G)$ or the smallest number of colors for which a k-coloring exists), R maximum allowed perturbations.
2: **Output**: H the smallest k-colorable subgraph
/*Initialization*/
3: $H_1 = \emptyset, \cdots, H_R = \emptyset$ /* each H_r records a detected candidate k-VCS*/
4: $A = \emptyset, B = V, C = \emptyset, H = (A, E_A)$ /*A is set of critical vertices, B is set of unknown vertices, C is set of uncritical vertices*/
5: $r = 0$ /*r is the number of performed perturbations*/
6: **while** $r < R$ **do**
7: **while** H does not contains a k-VCS **do**
8: Choose a vertex $i \in B$, move i from B to C /*vertex removal, Section 3.2*/
9: **if** $G_{A \cup B}$ does not contains a k-VCS **then**
10: move i from C to A /*i is detected as a critical vertex*/
11: $H = (A, E_A)$
/*Backtracking, Section 3.4*/
12: **while** $G_{A \cup B}$ does not contains a k-VCS **do**
13: Choose a vertex $l \in C$, move l from C to B
14: **end while**
15: **end if**
16: **end while**
17: $r = r + 1$
18: $H_r = H$
/*Perturbation operator for the critical subgraph*/
19: $H' \leftarrow Perturbation(H_r)$ /*Section 2*/
/*Update the set A, B and C*/
20: $\{A, B, C\} \leftarrow Update\_set(H')$ /*Section 3.6*/
21: $H = (A, E_A)$
22: **end while**
23: $H \leftarrow$ the smallest k-colorable subgraph

incorrectly classified into the uncritical set C, a *backtracking procedure* is invoked (line 12). This process is repeated until the subgraph induced by set A of critical vertices is such that its chromatic number equals k (line 7). One notices that the *removal procedure* is both greedy (guided by its vertex selection rule) and intensification-oriented (aiming to minimize the set of critical vertices). In order to diversify the search, we introduce two specific strategies based on the ideas of backtracking and perturbation.

2. Since the algorithm used to judge whether the remaining graphs $G_{A \cup B}$ contains a k-VCS is not an exact algorithm (see Section 3.3), a critical vertice can be incorrectly classified into the uncritical set C. To remedy this problem, we introduce a *backtracking procedure* which reconsiders the vertices of C (i.e., moving them back to the unknown set B, see Section 3.4). This procedure extends the removal procedure by continually moving uncritical vertices from C to B until the chromatic number of $G_{A \cup B}$ increases to k again (lines 12-14, see Section 3.4). Once $\chi(G_{A \cup B}) = k$ (which means the graph $G_{A \cup B}$ once again contains a k-VCS), the backtracking phase stops and resumes the removal phase.
3. The *perturbation procedure* is used to remedy the problem of some misclassified critical vertices. It moves vertices from set A to B (line 19, see

Section 2) and updates the sets A, B and C and the subgraph H (lines 20-21, see Section 3.6). Then, the removal algorithm is applied on the graph G with the updated sets. This phase terminates once the allowed number of perturbations is reached (line 6).

We observe that Algorithm 1 becomes the conventional removal algorithm of [5] when the backtrack procedure (lines 12-14) and perturbation procedure (lines 17-21) are disabled.

In the remainder of this section, we explain the main procedures of the IBR algorithm: the removal procedure, the heuristic coloring procedure, the backtracking procedure and the perturbation procedure.

3.2 The Removal Algorithm

Our removal procedure is based on the removal heuristic algorithm presented in [5], which can be conveniently described using the notion of critical, unknown and uncritical sets of vertices.

After initialization, $B = V$, $A = \emptyset$, $C = \emptyset$, $H = (A, E_A)$. We note that $G_{A \cup B} = G$ is not $(k-1)$-colorable, thus contains a k-VCS (see Section 3.1 and Algorithm 1). In each iteration, the removal algorithm tentatively moves one vertex i from the unknown set B to the uncritical set C. If $G_{A \cup B}$ without vertex i is always not $(k-1)$-colorable (i.e., $G_{A \cup B}$ contains a k-VCS), the removed vertex i is considered as an *uncritical* vertex and is kept in C. Otherwise, if $G_{A \cup B}$ without vertex i becomes $(k-1)$-coloring (i.e., $G_{A \cup B}$ does not contain a k-VCS), vertex i is considered as a *critical* vertex and thus transferred from set C to set A. In both cases, we update the sets A, B, C, the graphs H, $G_{A \cup B}$ and the weights of vertices accordingly. The removal algorithm repeats this process until $\chi(H)$ increases to k, which occurs when a k-VCS is detected.

In the removal algorithm, the rule used to select the next vertex from B impacts the critical subgraphs generated. In order to favor small critical subgraphs, the removal algorithm adopts the neighborhood weight heuristic strategy proposed in [5], which uses dynamically learned information contained in the weights of vertices.

Recall that when a vertex i is placed in the set A, B and C, its weight is set as $|E|$, 1 and 0 correspondingly. The neighborhood weight $W(i)$ of a vertex i is defined as the sum of the weights of the vertices that are adjacent to this vertex. When all weights of vertices are equal to 1 (e.g., after initialization), the neighborhood weight of a vertex equals its degree and can be considered as an indicator on the density of the region surrounding this vertex. The neighborhood weight heuristic strategy moves the vertices with the smallest neighborhood weight firstly, and preserves the denser ones. This strategy proved to be effective to detect a k-VCS with a small size [5].

3.3 Heuristic Coloring Algorithm

We adopt a heuristic coloring algorithm *Color* to judge whether a graph G is $(k-1)$-colorable (i.e., whether G contains a k-VCS). The removal algorithm can guarantee that the extracted subgraph $H = (A, E_A)$ is a k-VCS when the coloring algorithm *Color* is an *exact* algorithm (i.e., able to verify that the given remaining graph is colorable with a given k). However, given that the general k-coloring problem is itself NP-complete, exact methods can be too time consuming even for graphs of relatively small sizes (with several tens of vertices). For this reason and like [5], we adopt a heuristic coloring algorithm (i.e., TabuCol [17,7,9]). Contrary to an exact algorithm, the heuristic algorithm may fail to find a legal coloring with a given k, even if such a coloring exists.

Recall that at each iteration of the removal procedure, we move one unknown vertices i from B to C. Then, one needs to know whether there is a legal $(k-1)$-coloring on graph $H = (A, E_A)$ (the stopping condition is met or not) and on graph $G_{A\cup B} = (A \cup B, E_{A\cup B})$ (the last moved vertex is critical or not).

To solve the $(k-1)$-coloring problem, TabuCol iteratively explores partitions of V in $k-1$ classes. Each partition c is attributed to a fitness value $f(c)$ which is the number of edges of the graph that have both endpoints in the same class. Therefore, if $f(c) = 0$, the partition c corresponds to a proper $(k-1)$-coloring. The purpose of the tabu search coloring algorithm is then to minimize the fitness function f. The tabu coloring process stops when $f(c) = 0$ or when the fitness function cannot be improved within a given number of iterations.

3.4 Backtrack-based Removal Approach

The purpose of the backtracking phase is to cope with the problem caused by the heuristic coloring algorithm, that some critical vertices may be mistakenly moved to C. This situation happens as follows. When a *critical* vertex i is moved from B to C, *Color* should find a legal $(k-1)$-coloring in $G_{A\cup B}$ with $f_{A\cup B} = 0$ (Algorithm 1, line 9), and the vertex i should be moved from C to A (Algorithm 1, line 10). However, since our coloring algorithm is not an exact method, it may fail to find a $(k-1)$-coloring in $G_{A\cup B}$. In this circumstance, the critical vertex is unexpectedly classified to set C and $\chi(G_{A\cup B})$ decreases to $k-1$. To alleviate this problem, we calls for a backtracking scheme. When we find a legal $(k-1)$-coloring for graph $G_{A\cup B}$, we invoke the backtrack procedure, i.e., we move vertices from C to set B, until $\chi(G_{A\cup B})$ increases to k again. The key issue concerns the way to select vertices of C that are moved back to the unknown set B.

To make the choice, we consider two strategies: (1) according to the reverse order by which the vertices have been moved from the set B to the set C; (2) according to the non-increasing order of neighborhood weights (see

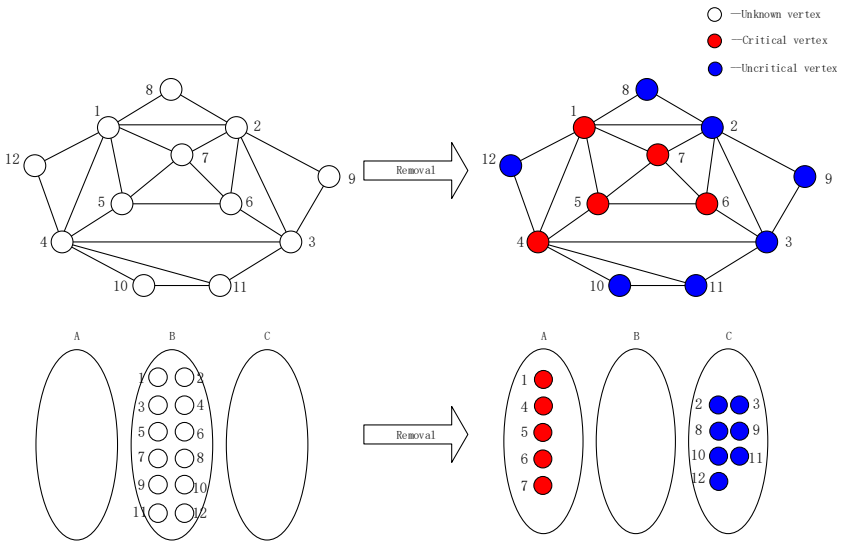


Fig. 3 Illustration of the removal procedure.

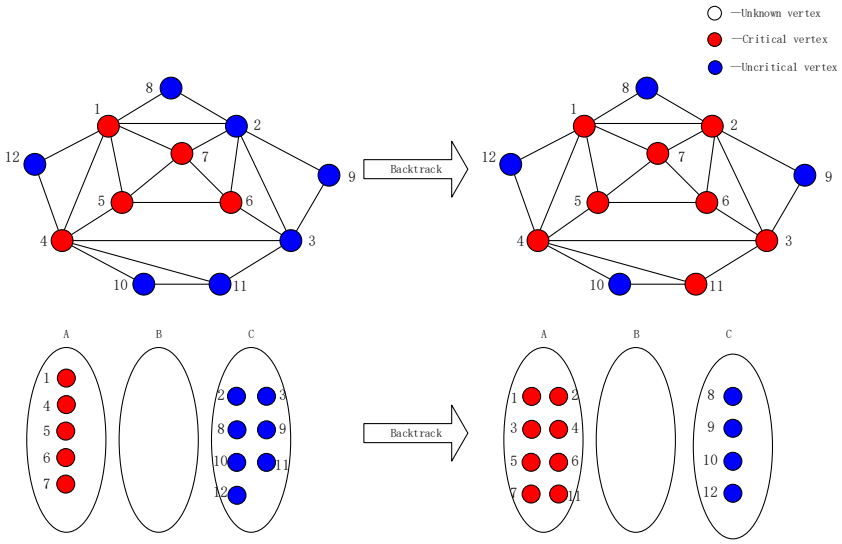


Fig. 4 Illustration of the backtrack-based removal procedure.

Section 3.2). Experiments indicated that the first strategy performs quite well and is thus used in the paper (see Section 5.3 for a computational analysis). This can be explained by the removal heuristic algorithm which prefers vertices with a small neighborhood weight. Thus, the last moved vertex (with a larger sum of neighborhood weights) has more chance to be a critical vertex compared to a vertex that was moved since long time. While in rare cases, there are also erroneous movements such that some uncritical vertices are moved to the critical set A.

To illustrate the procedure, we consider the graph G of Figure 1(a) ($\chi(G) = 4$) and the 4-VCS shown in Figure 1(b) (with vertices $\{1, 2, 3, 4, 5, 6, 7\}$). Figure 3 shows a possible situation in the removal procedure: suppose that the coloring algorithm *Color* fails to find a legal 3-coloring for $G_{A\cup B}$ when the vertices 2, 3 were moved to the uncritical set C, then the vertices 2, 3 are retained in the set C. In this case, $A = \{1, 4, 5, 6, 7\}$, $B = \emptyset$, $C = \{2, 3, 8, 9, 10, 11, 12\}$. Suppose that after adding back the last removed vertex 6 to the graph $G_{A\cup B}$, we detected that $G_{A\cup B}$ becomes 3-colorable ($G_{A\cup B}$ does not contain a 4-VCS) using the heuristic coloring algorithm, the backtracking phase is invoked. This involves moving vertices from set C to set A according to the reverse order by which the vertices have been moved from set B to set C, until $\chi(G_{A\cup B})$ increases to 4 again ($G_{A\cup B}$ contains a 4-VCS again). Figure 4 illustrates this backtracking procedure for the given graph G. However, if the uncritical vertex 11 is moved before the last critical vertex was moved, the uncritical vertex 11 will also be moved into the critical set A. Thus $A = \{1, 2, 3, 4, 5, 6, 7, 11\}$, $B = \emptyset$, $C = \{8, 9, 10, 12\}$. We note that this backtracking procedure is effective for repairing the misjudgments that misclassify critical vertices as uncritical vertices. Meanwhile, this procedure may unfortunately classify uncritical vertices into the critical set A.

3.5 Perturbation Operator of Backtrack-based Removal algorithm

As illustrated in Section 3, the backtrack-based removal procedure ends up with a k-VCS (Algorithm 1, lines 6-16). However, it may happen that some uncritical vertices are misclassified as critical vertices (always due to the use of a heuristic coloring algorithm). To overcome this problem, we introduce a perturbation procedure, which partially reconfigures A, B and C by moving some vertices from set A to set B. This procedure also provides new opportunities of finding k-VCS of smaller sizes.

The perturbation procedure (Algorithm 2) is inspired by the techniques proposed by [22,14], which is decomposed into two parts:

1. Scoring the vertices in set A. We define a function $score(i)$ to score each vertex i. Let $FlipFreq(i)$ be the number of times vertex i has been displaced among sets A, B and C, $EliteSol = [H_1, \cdots, H_R]$ a set of R critical subgraphs discovered so far (R is the size of $EliteSol$), $EliteFreq(i)$ the total number of times vertex i appears in $EliteSol$, the scoring function is then defined as:

Algorithm 2 Perturbation

Input: a k-critical-vertex-subgraph H, a probability threshold p
Output: a perturbed solution H' /*Score the vertices in the set A*/
for $i \in A$ **do**
  calculate $score(i)$ with Equation 2
end for
$A^{'} \leftarrow \eta$ vertices with the highest scores
$A^{'} \leftarrow$ sort $A^{'}$ in descending order of the scores /*Choose and move the perturbed vertices*/
for $i \in A^{'}$ **do**
  i is the jth element in $A^{'}$, calculate the possibility P_j using Equation 3
  if $P_j > p$ **then**
    move i from A to B
  end if
end for
$H' = (A, E_A)$

$$sorce(i) = EliteFreq(i)(l - EliteFreq(i))/r^2 + (1 - FliteFreq(i)/Max\_Freq) \tag{2}$$

where $1 \leq r \leq R$ and $Max\_Freq = max_{i \in \{1...,|V|\}}\{FlipFreq(i)\}$.

2. Choosing and moving the perturbed vertices. We sort all vertices in non-increasing order according to their scores and then choose probabilistically certain vertices from set A. The possibility of the jth highly-scored vertex being selected is given by:

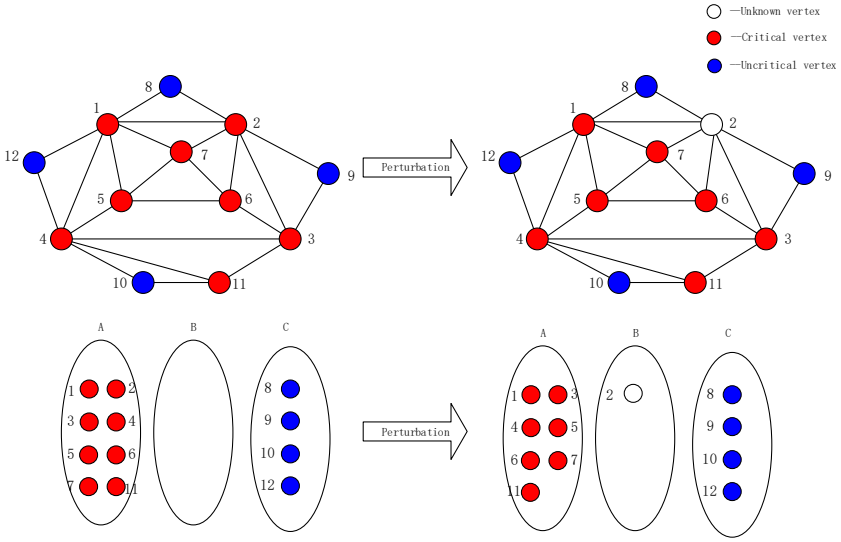
$$P_j = \frac{j^{-1}}{\sum_{z=1}^{\eta} z^{-1}} \tag{3}$$

where η is usually set as $|A|/2$. Then we move the chosen vertices from set A to set B and update the sets A, B and C (see Section 3).

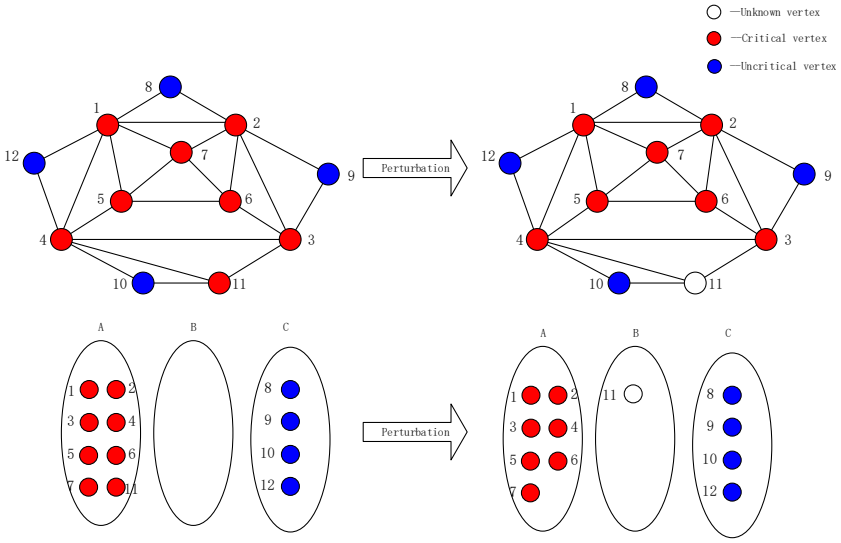
As an example, we consider the graph in Figure 5 with a chromatic number of 4 and one 4-VCS, i.e., $\{1, 2, 3, 4, 5, 6, 7\}$. Following Section 3.4, after the backtrack-based removal procedure, $A = \{1, 2, 3, 4, 5, 6, 7, 11\}$ as critical vertices and other ones are uncritical vertices ($C = \{8, 9, 10, 12\}$, $B = \emptyset$}), while the vertex 11 is mistaken for a critical vertex. The perturbation procedure moves certain vertices from set A to set B according to Algorithm 2. Figure 5(a) and Figure 5(b) respectively present two situations where the uncritical vertex 2 and the uncritical vertex 11 are moved from the set A to the set B by the perturbation procedure.

3.6 Update Procedure

Before invoking a next round of the backtrack-based removal procedure after the perturbation operation, an additional update operation is applied



(a) Case 1 for the perturbation procedure.



(b) Case 2 for the perturbation procedure.

Fig. 5 Illustration of the perturbation procedure.

Algorithm 3 Update_Set

1: **Input**: a perturbed graph H' ;
2: **Output**: updated sets A, B, C
3: $f_{H'} = Color(H', k-1)$
4: **if** $f_{H'} = 0$ **then**
5: move all vertices from C to B /*in order to re-detect each vertex of $(B \cup C)$ is uncritical or not*/
6: **else**
7: move all vertices from B to C /*all the vertices in set B are uncritical vertices*/
8: move all vertices from A to B /*in order to rejudge each vertex of A is critical or not*/
9: **end if**

(Algorithm 3). According to whether the perturbed solution contains a k-VCS, we use different strategies to update sets A, B and C. If the subgraph $H' = (A, E_A)$ does not contain a k-VCS (see Algorithm 3, line 4), which means certain critical vertices are misclassified in set B or even in set C, we move all vertices of C to B in order to re-examine each vertex of $B \cup C$ in the next step. Otherwise, we come to the conclusion that all vertices in set B are uncritical and move all vertices from B to C. Then, in order to determine whether the vertices of set A are uncritical or not, we move all vertices of A to B.

In Figure 5, after the backtracking procedure and the perturbation procedure, two cases are possible according to whether the subgraph $H' = (A, E_A)$ contains a 4-VCS. As shown in Figure 6(a), $H' = (A, E_A)$ does not contain a k-VCS. Then the update procedure moves all vertices from C to B in order to re-consider each vertex in $B \cup C$ in the next step. Accordingly, $A = \{1, 3, 4, 5, 6, 7, 11\}$, $B = \{2, 8, 9, 10, 12\}$ and $C = \emptyset$. Figure 6(b) shows a subgraph $H' = (A, E_A)$ that contains a 4-VCS. In this case, the update procedure moves all vertices from B to C and all vertices from A to B, leading to $A = \emptyset$, $B = \{1, 2, 3, 4, 5, 6, 7\}$ and $C = \{8, 9, 10, 11, 12\}$.

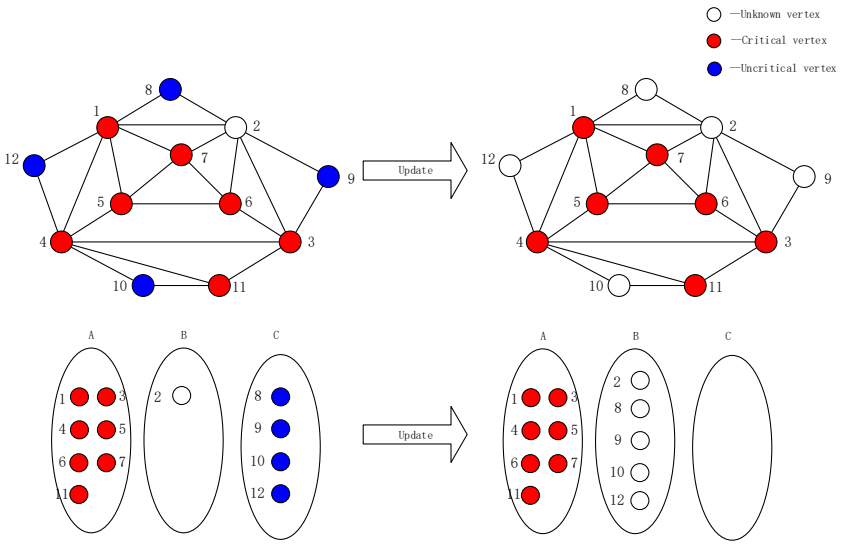
4 Experimental Results and Analysis

In this section, we assess the performance of the proposed IBR algorithm on a collection of benchmark graphs from the DIMACS[1] and COLOR02/03/04 competitions[2].

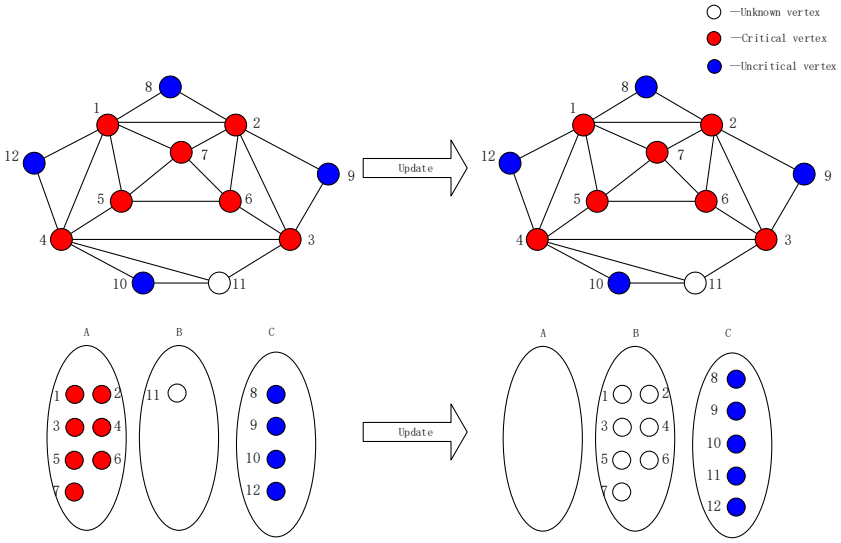
4.1 Experiment Settings

The proposed algorithm was programmed in C and compiled by GNU g++ with the -O3 flag (option). The experiments were conducted on a computer with Xeon E5440 (2.83GHz CPU and 2GB RAM) and Ubuntu Linux system

[1] http://www.dimacs.rutgers.edu/
[2] http://www.cs.hbg.psu.edu/txn131/graphcoloring.html/



(a) Case 1 of the update procedure.



(b) Case 2 of the update procedure.

Fig. 6 Illustration of the update procedure.

12.04. Running the DIMACS machine benchmark program dfmax.c[3], our computer requires 0.23, 1.42 and 5.42 seconds to solve graphs r300.5, r400.5, and r500.5, respectively.

For our comparative study, we used the best performing heuristic algorithm $Ins + h$ [5] as our main reference. The $Ins + h$ algorithm was run on an Athlon processor (1.6 GHz and 512 Mb of RAM). The comparison was performed mainly by considering the quality criterion of the solutions found. We note that our processor is about 1.8 times faster than that used by the reference algorithm. Thus, in all the experiments, our recorded CPU times were multiplied by 2 in order to make a reasonable comparison. Given that the compared algorithms were tested on different computing platforms and the runtime of an algorithm depends also on other factors (programming language, data structures, etc), it is difficult to strictly compare the runtimes. Thus, timing information was just provided for indicative purposes.

4.2 Instances and Experimental Settings

Test instances. We considered the set of 80 popular benchamrk graphs which were tested in [5]. These graphs are divided into four categories.

1. The first category contains 17 instances that are probably critical themselves. Graphs having *myciel**, *mug** or *Insertions** in their name fall into this category. The reference algorithm ($Ins + h$) extracts the minimum k-VCS for all graphs in this category at $k = \chi(G)$.
2. The second category contains instances which have cliques as minimum k-VCS, for $k = \chi(G)$. This category of graphs includes *le*450*, *fpso**, *inithx**, *mulsol**, *zeroin**, *school*1*, *miles**, *anna*, *david*, *homer*, *huck*, *jean* graphs. The $Ins + h$ algorithm also detects the optimum k-VCS at $k = \chi(G)$ for all graphs in this category.
3. The third category contains the instances which have DSJC in their name. According to [5], these graphs are harder than other graphs for which a k-VCS is difficult to detect. In fact, the $Ins + h$ algorithm fails to find a k-VCS for most of graphs in this category.
4. The last category is composed of all the instances falling in none of the above three categories.

Parameters. Following [5], we ran our IBR algorithm 5 times to solve reach instance, and reported the best values of the successful runs.

Stopping condition. The IBR algorithm stops once the number of the perturbation reaches the given limit $R = 20$.

Quality criteria. The experiments have two goals. The first one is to compare the quality of the detected k-VCS for the given k. In this case, a smaller k-VCS represents a better solution. The second goal concerns the quality of the lower bound of $\chi(G)$ that an algorithm can reach [5]. In this case, we try

[3] dfmax: ftp://dimacs.rutgers.edu/pub/dsj/clique/

Table 1 Comparative results of IBR with state-of-the-art algorithm on the first category of instances.

Instance				$Ins+h$ [5]				IBR			
name	$\|V\|$	$\|E\|$	k_UB	k	$\|V\|$	$\|E\|$	$time(s)$	k	$\|V\|$	$\|E\|$	$time(s)$
myciel5	47	236	6*	6*	47*	236*	0.1	6*	47*	236*	0.00
myciel6	95	755	7*	7*	95*	755*	0.2	7	95	755	0.01
myciel7	191	2360	8*	8*	191*	2360*	30.65	8*	191*	2360*	11.46
mug88_1	88	146	4*	4*	88*	146*	0.05	4*	88*	146*	0.03
mug88_25	88	146	4*	4*	88*	146*	0.05	4*	88*	146*	0.02
mug100_1	100	166	4*	4*	100*	166*	0.05	4*	100*	166*	0.03
mug100_25	100	166	4*	4*	100*	166	0.05	4*	100*	166*	0.03
1_Insertion_4	67	232	5*	5*	67*	232*	0.05	5*	67*	232*	0.3
1_Insertion_5	202	1227	6	6	202	1227	0.05	6	202	1227	0.23
1_Insertion_6	607	6337	7	7	607	6337	49.85	7	607	6337	390.24
2_Insertion_4	149	541	5	5	149	541	0.15	5	149	541	0.04
2_Insertion_5	597	3936	6	6	597	3936	8.0	6	597	3936	14.67
3_Insertion_3	56	110	4*	4*	56*	110*	0.2	4	56*	110*	0.02
3_Insertion_4	281	1046	5	5	281	1046	0.4	5	281	1046	0.58
3_Insertion_5	1406	96,957	6	6	1406	96, 957	257.4	6	1406	96, 957	1580.37
4_Insertion_3	79	156	4	4	79	156	0.3	4	79	156	0.02
4_Insertion_4	475	1795	5	5	475	1795	0.75	5	475	1795	0.95

to find k-VCS with increasing k values (so a k-VCS with a larger k is better, which corresponds to a tighter lower bound of $\chi(G)$).

Finally, like [5], we verify the validity of a candidate k-VCS returned by the IBR algorithm with an exact coloring algorithm. Indeed, since the returned k-VCS is usually of small size, its chromatic number can be determined exactly by a modern exact coloring algorithm within a time frame of several hours. In our case, we used the recent algorithm presented in [29].

4.3 Comparison with State of the Art Algorithm

Tables 1, 2, 3 and 4 summarize the computational results of our IBR algorithm on the four categories of DIMACS benchmark graphs, respectively. In these tables, the first 4 columns show the name of each instance, the number of vertices V, the number of edges E and the best-known upper bound k_UB of the chromatic number reported in the literature. Values followed by an asterisk $*$ indicate that they correspond to $\chi(G)$. The following four columns indicate the results of the $Ins+h$ algorithm [5]: the k value for which a k-VCS is detected (i.e., lower bound of $\chi(G)$), the number of vertices and the number of edges of the detected k-VCS, and the CPU time in seconds needed to find the k-VCS. The last four columns show the results obtained by our IBR algorithm. In Table 3, we show some improved lower bounds obtained by the IBR algorithm compared to the lower bounds obtained by the $Ins+h$ algorithm. For each of these cases, we list the best lower bound that IBR can achieve and its corresponding k-VCS in an additional row.

Table 1 displays the results of $Ins+h$ and the results of IBR for the 17 instances of the first category, which are probably critical graphs themselves [5]. Both algorithms can obtain the optimal subgraphs, but IBR is faster than $Ins+h$ for 10 graphs. The same observation can be made for the 39 instances

of the second category which have maximum cliques as minimum k-VCS for $k = \chi(G)$.

Table 2: Comparative results of IBR with state-of-the-art algorithm on the second category of instances.

Instance				$Ins + h$ [5]				IBR			
name	$\|V\|$	$\|E\|$	k_UB	k	$\|V\|$	$\|E\|$	$time(s)$	k	$\|V\|$	$\|E\|$	$time(s)$
le450_5a	450	5714	5*	5*	5*	10*	5.35	5*	5*	10*	2.24
le450_5b	450	5734	5*	5*	5*	10*	7.7	5*	5*	10*	2.48
le450_5c	450	9803	5*	5*	5*	10*	8.9	5*	5*	10*	2.33
le450_5d	450	9757	5*	5*	5*	10*	8.35	5*	5*	10*	3.67
le450_15a	450	8168	15*	15*	15*	105*	5.4	15*	15*	105*	1.39
le450_15b	450	8169	15*	15*	15*	105*	3.0	15*	15*	105*	1.4
le450_15c	450	16680	15*	15*	15*	105*	22.25	15*	15*	105*	6.8
le450_15d	450	16750	15*	15*	15*	105*	14.65	15*	15*	105*	5.39
le450_25a	450	8260	25*	25*	25*	300*	7.2	25*	25*	300*	3.34
le450_25b	450	8263	25*	25*	25*	300*	6.6	25*	25*	300*	3.14
le450_25c	450	17343	25*	25*	25*	300*	9.1	25*	25*	300*	4.07
le450_25d	450	17425	25*	25*	25*	300*	8.95	25*	25*	300*	6.65
school1	385	19,095	14*	14*	14*	91*	6.25	14*	14*	91*	1.94
school1_nsh	385	19,095	14*	14*	14*	91*	15.1	14*	14*	91*	1.19
miles250	128	387	8*	8*	8*	28*	0.1	8*	8*	28*	0.04
miles500	128	1170	20*	20*	20*	190*	0.1	20*	20*	190*	0.09
miles750	128	2113	31*	31*	31*	465*	1.05	31*	31*	465*	0.41
miles1000	128	3216	42*	42*	42*	861*	1.5	42*	42*	861*	1.33
miles1500	128	5198	73*	73*	73*	2628*	1.6	73*	73*	2628*	0.64
anna	138	493	11*	11*	11*	55*	0.15	11*	11*	55*	0.07
david	87	406	11*	11*	11*	55*	0.15	11*	11*	55*	0.04
homer	561	1629	13*	13*	13	78*	0.4	13*	13*	78*	0.23
huck	74	301	11*	11*	11*	55*	0.1	11*	11*	55*	0.04
jean	80	254	10*	10*	10*	45*	7.1	10	10*	45*	0.09
games120	120	638	9*	9*	9*	36*	0.2	9*	9*	36*	0.07
fpsol2.i.1	496	11654	65*	65*	65*	2080*	104.95	65*	65*	2080*	21.93
fpsol2.i.2	451	8691	30*	30*	30*	435*	10.0	30*	30*	435*	3.33
fpsol2.i.3	451	8691	30*	30*	30*	435*	26.25	30*	30*	435*	4.34
mulsol.i.1	197	3925	49*	49*	49*	1176*	2.2	49*	49*	1176*	0.47
mulsol.i.2	188	3885	31*	31*	31*	465*	3.2	31*	31*	465*	0.5
mulsol.i.3	184	3916	31*	31*	31*	465*	3.25	31*	31*	465*	0.47
mulsol.i.4	185	3946	31*	31*	31*	465*	3.4	31*	31*	465*	0.37
mulsol.i.5	186	3973	31*	31*	31*	465*	3.45	31*	31*	465*	1.86
zeroin.i.1	211	4100	49*	49*	49*	1176*	2.2	49	49*	1176*	0.45
zeroin.i.2	211	3541	30*	30*	30*	435*	5.7	30*	30*	435*	1.69
zeroin.i.3	206	3540	30*	30*	30*	435*	2.8	30*	30*	435*	1.68
inithx.i.1	864	18707	54*	54*	54*	1431*	448.85	54*	54*	1431*	38.13
inithx.i.2	645	13979	31*	31*	31*	465*	5.1	31	31*	465*	0.99
inithx.i.3	621	13969	31*	31*	31*	465*	7.1	31*	31*	465*	1.9

The most interesting results concern the 8 random instances of the third category. When comparing IBR and $Ins + h$ for this category, one observes that IBR improves the lower bound for 6 instances (DSJC125.5, DSJC250.1, DSJC250.5, DSJC500.1, DSJR500.1c, DSJR500.5). Moreover, at the same k, IBR obtains a better solution (smaller size of k-VCS) for 5 instances (DSJC125.5, DSJC250.1, DSJC250.5, DSJC500.1, DSJR500.1c). IBR is also faster than $Ins + h$ on all instances when using the same k.

The results on the instances of the fourth category are shown in Table 4. One observes that IBR improves the best-known results for 3 instances while matching the best-known results for other 12 instances. Only in one case, IBR obtains a worse result.

Table 3 Comparative results of IBR with state-of-the-art algorithm on the third category of instances. Improved results are indicated in bold.

Instance				*Ins* + *h* [5]				IBR			
name	$\|V\|$	$\|E\|$	*k_UB*	*k*	$\|V\|$	$\|E\|$	*time*(*s*)	*k*	$\|V\|$	$\|E\|$	*time*(*s*)
DSJC125.1	125	736	5*	5*	10	26	0.4	5*	10	26	2.29
DSJC125.5	125	3891	17*	14	70	1341	46.35	14	**66**	**1266**	11.69
								15	**82**	**1862**	35.4
DSJC250.1	250	3218	8	6	64	362	27.65	6	**51**	**290**	21.45
								7	**120**	**983**	134
DSJC250.5	250	15,668	28	14	74	1505	59.6	14	**50**	**836**	32.54
								16	**75**	**1742**	43.92
DSJC500.1	500	12,458	12*	6	65	369	73.15	6	**32**	**159**	48.23
								7	**79**	**617**	353.06
DSJR500.1	500	3555	12*	12*	12*	66*	1.9	12*	12*	66*	41.23
DSJR500.1C	500	121,275	85*	80	84	3477	710.75	80	**80**	**3160**	153.96
								83	**83**	**3403**	1280.19
DSJR500.5	500	58,862	122	90	90	4005	373.6	90	90	4005	15.78
								119	**119**	**7,021**	29.08

Table 4 Comparative results of IBR with state-of-the-art algorithm on the fourth category of instances. Improved results are indicated in bold.

Instance				*Ins* + *h* [5]				IBR			
name	$\|V\|$	$\|E\|$	*k_UB*	*k*	$\|V\|$	$\|E\|$	*time*(*s*)	*k*	$\|V\|$	$\|E\|$	*time*(*s*)
queen6_6	36	290	7*	7*	**22**	**119**	0.8	7*	24	140	1.24
queen8_8	64	728	9*	9*	54	538	12.6	9*	**53**	**519**	2.6
queen9_9	81	2112	10*	10*	74	897	13.8	10*	**72**	**869**	920.45
ash331GPIA	662	4185	4	4	9	16	1.6	4	**7**	**12**	287.08
1-FullIns_3	30	100	4	4	7	12	0.1	4	7	12	0.09
1-FullIns_4	93	593	5	5	15	43	**0.25**	5	15	43	0.32
1-FullIns_5	282	3247	6	6	31	144	7.3	6	31	144	2.10
2-FullIns_3	52	201	5	5	9	22	0.1	5	9	22	0.09
2-FullIns_4	212	1621	6	6	19	75	**0.25**	6	19	75	0.58
2-FullIns_5	852	12,201	7	7	39	244	**13.3**	7	39	244	22.4
3-FullIns_3	80	346	6	6	11	35	0.15	6	11	35	0.15
3-FullIns_4	405	3524	7	7	23	116	**1.25**	7	23	116	3.38
3-FullIns_5	2030	33,751	8	8	47	371	**78.6**	8	47	371	396.26
4-FullIns_3	114	541	7	7	13	51	0.2	7	13	51	0.25
4-FullIns_4	690	6650	8	8	27	166	**12.3**	8	27	166	15.35
5-FullIns_3	154	792	8	8	15	70	1.2	8	15	70	0.47

5 Analysis and Discussions

This section performs additional experiments to analyze the proposed IBR algorithm: the backtrack strategy and the perturbation operator. These experiments were performed on a selection of 13 representative instances.

5.1 Effectiveness of Different Backtrack Strategies for *k*-VCS Detection

This section compares the backtracking strategy adopted by IBR (i.e., using the reverse order by which the vertices were moved from the set B to the set C) and the backtracking rule according to the non-increase order of the sum of neighborhood weights of the vertices in the current graph. We use IBR1 to

denote the IBR variant using the second backtracking strategy. We ran both IBR and IBR1 5 times to obtain the k-VCS of each instance.

Table 5 Comparative results of the IBR algorithm with two different backtrack strategies. Improved results are indicated in bold.

Instance				IBR1					IBR				
name	$\|V\|$	$\|E\|$	k_UB	k	$\|V\|$	$\|E\|$	$time(s)$	SR	k	$\|V\|$	$\|E\|$	$time(s)$	SR
DSJC125.5	125	3891	17	15	83	1,898	1.3	2/5	15	**82**	**1854**	35.4	5/5
DSJC250.1	250	3218	8	7	**116**	995	30.57	1/5	7	120	**983**	133.56	1/5
DSJC250.5	250	15,668	28	16	77	1675	73.37	1/5	16	**75**	**1742**	43.92	1/5
DSJC500.1	500	12,458	12	7	83	671	276.58	1/5	7	**79**	**617**	353.06	1/5
DSJR500.1C	500	121,275	85	83	83	3403	1255.93	1/5	83	83	3403	1280.19	5/5
DSJR500.5	500	58,862	122	119	119	7,021	21.91	5/5	119	119	7,021	29.08	5/5
queen6_6	36	290	7	7	24	140	1.0	2/5	7	24	140	1.24	1/5
queen8_8	64	728	9	9	54	542	14.92	1/5	9	**53**	**519**	2.6	1/5
queen9_9	81	2112	10	9	9	36	0.35	2/5	9	9	36	0.35	5/5
									10	**72**	**869**	920.45	5/5
ash331GPIA	662	4185	4	4	7	12	132.55	3/5	4	7	12	287.08	2/5
1-FullIns_5	282	3247	6	6	31	144	3.07	1/5	6	31	144	2.10	2/5
2-FullIns_5	852	12,201	7	7	39	244	53.07	2/5	7	39	244	22.4	5/5
3-FullIns_5	2030	33,751	8	8	48	374	800.85	1/5	8	**47**	**371**	396.26	2/5

The experimental results are presented in Table 5, including the lower bound k, the number of vertices and edges of the k-VCS found, and the success rate (SR) to obtain the k-VCS over 5 runs. When comparing the k-VCS obtained by IBR and IBR1, one observes that IBR obtains better solutions for 5 instances at the same k and improves the lower bound for 1 instance. This justifies the backtracking strategy used in our previous experiments.

5.2 Effectiveness of Perturbation for k-VCS Detection

As shown in Section 2, the proposed algorithm uses a perturbation strategy to reconfigure the sets A, B and C. In order to show the effect of the perturbation procedure, we compare IBR with a traditional restart strategy (denoted as BR) where each restart begins its search with an initial configuration $A = \emptyset, B = V, C = \emptyset, H = (A, E_A)$. The two algorithms were run 5 times on the 13 selected instances and the results are provided in Table 6.

From Table 6, we observe that IBR significantly outperforms BR. IBR dominates BR by finding improved lower bounds for two instances, smaller k-VCS sizes for 6 instances at the same k and no worse result. This experiment confirms the interest of the adopted perturbation operator.

5.3 Effectiveness of Backtrack for k-VCS Detection

This section evaluates the influence of the backtracking scheme on the performance of the proposed algorithm. For this purpose, we compare it with an IBR variant without the backtracking strategy (named IR). We ran both

Table 6 Analysis of the influence of the perturbation on the performance of the IBR algorithm. The BR algorithm is obtained by replacing the perturbation procedure of the IBR algorithm with a restart strategy. Improved results are indicated in bold.

Instance				BR					IBR				
name	$\|V\|$	$\|E\|$	*k_UB*	*k*	$\|V\|$	$\|E\|$	*time*(*s*)	*SR*	*k*	$\|V\|$	$\|E\|$	*time*(*s*)	*SR*
DSJC125.5	125	3891	17	15	82	1858	24.98	1/5	15	82	**1854**	35.4	2/5
DSJC250.1	250	3218	8	7	116	1010	51.2	1/5	7	**120**	**983**	133.56	3/5
DSJC250.5	250	15,668	28	16	75	1740	42.75	1/5	16	**75**	**1740**	42.75	1/5
DSJC500.1	500	12,458	12	7	91	767	318.41	1/5	7	**79**	**617**	353.06	1/5
DSJR500.1C	500	121,275	85	80	80	3160	153.96	1/5	**80**	**80**	**3160**	153.96	5/5
									83	**83**	**3403**	1280.19	5/5
DSJR500.5	500	58,862	122	119	119	7,021	46.56	5/5	119	119	7,021	57.1	5/5
queen6_6	36	290	7	7	24	140	1.03	2/5	7	24	140	1.24	3/5
queen8_8	64	728	9	9	54	542	14.92	1/5	9	**53**	**519**	2.6	1/5
queen9_9	81	2112	10	9	9	36	0.35	2/5	9	9	36	0.35	5/5
									10	**72**	**869**	920.45	5/5
ash331GPIA	662	4185	4	4	9	16	11.86	5/5	4	**7**	**12**	287.08	2/5
1-FullIns_5	282	3247	6	6	31	144	1.48	1/5	6	31	144	2.10	2/5
2-FullIns_5	852	12,201	7	7	39	244	27.89	1/5	7	39	244	22.4	5/5
3-FullIns_5	2030	33,751	8	8	47	371	833.14	1/5	8	47	371	396.26	2/5

Table 7 Influence of the backtracking procedure on the performance of the IBR algorithm. IR is obtained by disabling the backtracking procedure. Improved results are indicated in bold.

Instance				IR					IBR				
name	$\|V\|$	$\|E\|$	*k_UB*	*k*	$\|V\|$	$\|E\|$	*time*(*s*)	*SR*	*k*	$\|V\|$	$\|E\|$	*time*(*s*)	*SR*
DSJC125.5	125	3891	17	14	67	1299	21.22	1/5	14	**66**	**1266**	11.69	5/5
									15	**82**	**1854**	35.4	5/5
DSJC250.1	250	3218	8	6	57	335	8.23	1/5	6	**51**	**290**	21.45	1/5
									7	**120**	**983**	133.56	3/5
DSJC250.5	250	15,668	28	13	38	478	1095.26	1/5	13	**35**	**434**	947.51	1/5
									16	**75**	**1742**	43.92	1/5
DSJC500.1	500	12,458	12	6	39	196	73.5	1/5	**6**	**32**	**159**	48.23	3/5
									7	**79**	**617**	353.06	1/5
DSJR500.1C	500	121,275	85	80	80	3160	153.96	1/5	80	**80**	**3160**	153.96	5/5
									83	**83**	**3403**	1280.19	5/5
DSJR500.5	500	58,862	122	119	119	7,021	31.71	5/5	119	119	7,021	57.1	5/5
queen6_6	36	290	7	7	26	163	0.05	5/5	7	**24**	**140**	1.24	1/5
queen8_8	64	728	9	9	54	542	14.92	1/5	9	**53**	**519**	2.6	1/5
queen9_9	81	2112	10	9	9	36	0.35	2/5	9	9	36	0.35	5/5
									10	**72**	**869**	920.45	5/5
ash331GPIA	662	4185	4	4	9	16	3.52	5/5	4	**7**	**12**	287.08	2/5
1-FullIns_5	282	3247	6	6	40	220	2.29	1/5	6	**31**	**144**	2.10	2/5
2-FullIns_5	852	12,201	7	7	39	244	46.31	1/5	7	39	244	22.4	5/5
3-FullIns_5	2030	33,751	8	8	47	371	275.60	1/5	8	47	371	396.26	2/5

algorithms 5 times on the 13 selected instances. The experimental results are presented in Table 7. Compared with IR, IBR obtains smaller k-VCS for 8 instances at the same given k, 6 better lower bounds, and no worse result. This experiment confirms the value of the backtracking strategy.

6 Conclusion

This paper presented the interated backtracking removal search for identifying small k-vertex-critical subgraph in a general graph. This method extends the previous removal algorithm by integrating a backtracking strategy

and a perturbation procedure that complement the intensification-oriented removal approach. These extensions provide two complementary ways to alleviate the problem of misclassifying vertices caused by the use of a heuristic coloring algorithm.

We assessed the performance of the IBR algorithm on the set of 80 benchmark instances from DIMACS and COLOR competitions and presented comparative results with respect to the state-of-the-art results. The comparisons showed that IBR performs very well by discovering several improved best results (9 smaller sized k-VCS for a given k and 6 new lower bounds for unfixed k) and matching the best-known results for the remaining instances except one case. This study demonstrates the benefit of the backtracking scheme and the perturbation procedure for solving the k-VCS problem. These ideas could be beneficially applied to other problems of identifying irreducible inconsistent subsets as those indicated in the introduction of this work. Finally, the k-VCS problem considered in this work and other critical nodes problems like those studied in [27] share an interesting property that the "critical" nodes form a stable structure (or subgraph). This property could be advantageously explored by a learning-based search procedure using pattern mining techniques [28]. We are currently investigating this research direction.

Acknowledgment

We are grateful to Dr. Chumin Li for providing us with the code of [29]. Support for the first author of this work from the China Scholarship Council (2015-2019) is also acknowledged.

References

1. Nilotpal Chakravarti. Some results concerning post-infeasibility analysis. *European Journal of Operational Research*, 73(1):139-143, 1994.
2. John W. Chinneck. Minos (IIS): infeasibility analysis using minos. *Computers and Operations Research*, 21(1):1-9, 1994.
3. John W. Chinneck. Finding a useful subset of constraints for analysis in an infeasible linear program. *INFORMS Journal on Computing*, 9(2):164-174, 1997.
4. John W. Chinneck and Erik W Dravnieks. Locating minimal infeasible constraint sets in linear programs. *ORSA Journal on Computing*, 3(2):157-168, 1991.
5. Christian Desrosiers, Philippe Galinier, and Alain Hertz. Efficient algorithms for finding critical subgraphs. *Discrete Applied Mathematics*, 156(2):244-266, 2008.
6. Christian Desrosiers, Philippe Galinier, Alain Hertz, and Sandrine Paroz. Using heuristics to find minimal unsatisfiable subformulas in satisfiability problems. *Journal of Combinatorial Optimization*, 18(2):124-150, 2009.
7. Raphael Dorne and Jin-Kao Hao. A new genetic local search algorithm for graph coloring, *Lecture Notes in Computer Science* 1498, 745–754, 1998.
8. Carlos Eisenberg and Boi Faltings. Using the breakout algorithm to identify hard and unsolvable subproblems. *Lecture Notes in Computer Science* 2833:822-826, 2003
9. Philippe Galinier and Jin-Kao Hao. Hybrid evolutionary algorithms for graph coloring. *Journal of Combinatorial Optimization*, 3(4):379-397, 1999.
10. Philippe Galinier, Alain Hertz, and Nicolas Zufferey. An Adaptive Memory Algorithm for the k-colouring problem. *Discrete Applied Mathematics*, 156(2):267-279, 2008.

11. Michael R. Garey and David S. Johnson. Computers and intractability: a guide to the theory of NP-completeness. *San Francisco, LA: Freeman*, 58, 1979.
12. Fred Glover. Tabu search-part I. *ORSA Journal on computing*, 1(3):190-206, 1989.
13. Fred Glover. Tabu search-part II. *ORSA Journal on computing*, 2(1):4-32, 1990.
14. Fred Glover, Zhipeng Lü, and Jin-Kao Hao. Diversification-driven tabu search for unconstrained binary quadratic problems. *4OR: A Quarterly Journal of Operations Research*,8(3): 239-253, 2010.
15. Eric Grégoire, Bertrand Mazure, and Cedric Piette. On Finding Minimally Unsatisfiable Cores of CSPs. *International Journal on Artificial Intelligence Tools*, 17(4):745-763, 2008.
16. Francine Herrmann and Alain Hertz. Finding the chromatic number by means of critical graphs. *Journal of Experimental Algorithmics (JEA)*, 7:10, 2002.
17. Alain Hertz, Dominique de Werra. Using tabu search techniques for graph coloring, Computing 39 (4), 345–351, 1987.
18. David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: an experimental evaluation; part I, graph partitioning. *Operations Research*, 37(6):865-892, 1989.
19. Mark H. Liffiton and Karem A. Sakallah. On finding all minimally unsatisfiable subformulas. In *International Conference on Theory and Applications of Satisfiability Testing*, 173-186. Springer, 2005.
20. Mark H. Liffiton and Karem A. Sakallah. Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints. *Journal of Automated Reasoning*, 40(11):133, 2008.
21. J.N.M. van Loon. Irreducibly inconsistent systems of linear inequalities. *European Journal of Operational Research*, 8(3): 283-288, 1981.
22. Zhipeng Lü and Jin-Kao Hao. A critical element-guided perturbation strategy for iterated local search. In *Evolutionary Computation in Combinatorial Optimization*, 1-12. Springer, 2009.
23. Inês Lynce and João P. Marques Silva. *On Computing Minimum Unsatisfiable Cores*. Online Proceedings of The Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004), 10-13 May 2004, Vancouver, BC, Canada.
24. Maher Mneimneh, Inês Lynce, Zaher Andraus, João Marques-Silva, and Karem Sakallah. A branch-and-bound algorithm for extracting smallest minimal unsatisfiable formulas. *International Conference on Theory and Applications of Satisfiability Testing*, 467-474. Springer, 2005.
25. Yoonna Oh, Maher N Mneimneh, Zaher S Andraus, Karem A Sakallah, and Igor L Markov. Amuse: a minimally-unsatisfiable subformula extractor. In *Proceedings of the 41st Annual Design Automation Conference*, 518-523. ACM, 2004.
26. Mehrdad Tamiz, Simon J Mardle, and Dylan F Jones. Detecting iis in infeasible linear programmes using techniques from goal programming. *Computers and Operations Research*, 23(2):113-119, 1996.
27. Yangming Zhou, Jin-Kao Hao, and Fred Glover. Memetic search for identifying critical nodes in sparse graphs. arXiv:1705.04119, May 2017.
28. Yangming Zhou, Jin-Kao Hao, and Béatrice Duval. When data mining meets optimization: A case study on the quadratic assignment problem. arXiv:1708.05214, August 2017.
29. Zhaoyang Zhou, Chu Min Li, Chong Huang, and Ruchu Xu. An exact algorithm with learning for the graph coloring problem. *Computers and Operations Research* 51:282-301, 2014.