Effective and Efficient Dynamic Graph Coloring

Long Yuan[§], Lu Qin[‡], Xuemin Lin[§], Lijun Chang[†], and Wenjie Zhang[§]
[§] The University of New South Wales, Australia
[‡]Centre for Quantum Computation & Intelligent Systems, University of Technology, Sydney, Australia
[†] The University of Sydney, Australia
[§]{longyuan,lxue,zhangw}@cse.unsw.edu.au; [‡]lu.qin@uts.edu.au;
[†]Lijun.Chang@sydney.edu.au

ABSTRACT

Graph coloring is a fundamental graph problem that is widely applied in a variety of applications. The aim of graph coloring is to minimize the number of colors used to color the vertices in a graph such that no two incident vertices have the same color. Existing solutions for graph coloring mainly focus on computing a good coloring for a static graph. However, since many real-world graphs are highly dynamic, in this paper, we aim to incrementally maintain the graph coloring when the graph is dynamically updated. We target on two goals: high effectiveness and high efficiency. To achieve high effectiveness, we maintain the graph coloring in a way such that the coloring result is consistent with one of the best static graph coloring algorithms for large graphs. To achieve high efficiency, we investigate efficient incremental algorithms to update the graph coloring by exploring a small number of vertices. We design a color-propagation based algorithm which only explores the vertices within the 2-hop neighbors of the update-related and color-changed vertices. We then propose a novel color index to maintain some summary color information and, thus, bound the explored vertices within the neighbors of these vertices. Moreover, we derive some effective pruning rules to further reduce the number of propagated vertices. The experimental results demonstrate the high effectiveness and efficiency of our approach.

PVLDB Reference Format:
Long Yuan, Lu Qin, Xuemin Lin, Lijun Chang and Wenjie Zhang. Effective and Efficient Dynamic Graph Coloring. *PVLDB*, 11(3): 338 - 351, 2017.
DOI: 10.14778/3157794.3157802

1. INTRODUCTION

Graph coloring is one of the most fundamental problems in graph analysis. Given a graph G, graph coloring assigns each vertex in G a color, such that no two incident vertices have the same color. The aim of graph coloring is to minimize the number of different colors. Computing the optimal graph coloring is an NP-hard problem [27].

Applications. Graph coloring has been adopted in a wide range of application scenarios. For example:

(1) Nucleic Acid Sequence Design in Biochemical Networks. Given a set of nucleic acids, a dependency graph is a graph in which each vertex is a nucleotide and two vertices are connected if the two nucleotides form a base pair in at least one of the nucleic acids. The problem of finding a nucleic acid sequence that is compatible with the set of nucleic acids can be modelled as a graph coloring problem on a dependency graph [57].

(2) Air Traffic Flow Management. In air traffic flow management, the air traffic flow can be considered as a graph in which each vertex represents a flight route and there is an edge between two vertices if the corresponding two routes intersect. The airspace congestion problem can be modelled as a graph coloring problem [6].

(3) Channel Assignment in Wireless Networks. In a wireless network, each device is represented as a vertex, and the potential interference between two devices is represented as an edge. The channel assignment aims to to cover all devices (vertices) with the minimum number of channels (colors) such that no two adjacent devices (vertices) use the same channel (color), which can be modelled as a graph coloring problem [5].

(4) Community Detection in Social Networks. In a social network, graph coloring is used to compute seed vertices that can be expanded to high quality overlapping communities in the network [49].

(5) A Key Step to Solve other Graph Problems. Graph coloring also serves as a key step to solve other important graph problems, such as clique computation [55, 70] and graph partitioning [3].

Motivation. Plenty of algorithms that handle the graph coloring problem in a static graph have been proposed, such as [7, 34, 40, 42, 57, 68]. However, many real-world graphs are highly dynamic [2, 19, 36], which raises the following two requirements for the graph coloring algorithms in this new scenario:

(1) Effectiveness. In dynamic graph coloring, besides minimizing the number of used colors [60], coloring consistency is also an important issue to be considered in real applications. Here, by consistency, we mean that the coloring result of the same graph is independent of the graph updating orders. For example, in channel assignment [5], power consumption is critical to the usability of mobile devices [14] and WiFi is a prime source of energy consumption [47]. Consistent coloring result can save the power of mobile devices by avoiding unnecessary channel assignments triggered by the movement of other mobile devices. In graph partitioning, consistent coloring result can reduce the repartitioning costs when the graph is updated, as graph coloring is used to classify vertices into different groups based on the colors of vertices [3].

(2) Efficiency. In real applications, many graphs are large and frequently updated. For example, in wireless networks, the access devices are frequently inserted/removed because of the movement of

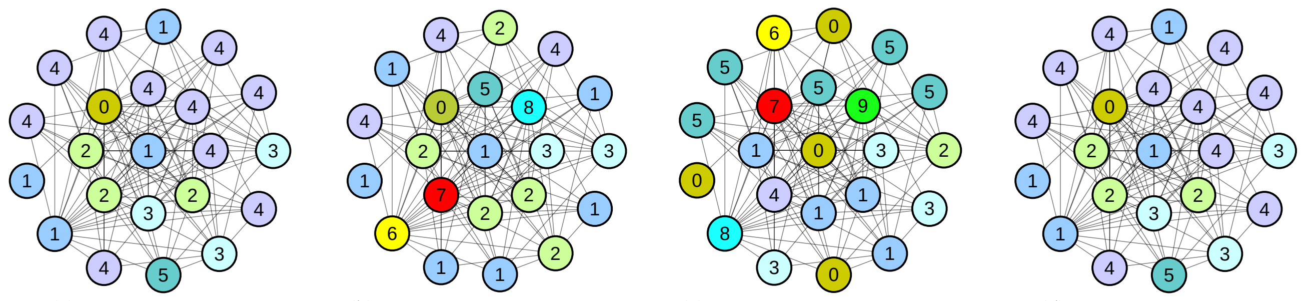

(a) The initial coloring (b) Color by DC-Local (1st test) (c) Color by DC-Local (2nd test) (d) Color by our algorithm

Figure 1: Graph coloring on part of the rating network *MoiveLens* (the numbers denote different colors)

people [62]; the air traffic flow in air traffic flow networks changes as flights are delayed or cancelled [51]; In the online social networks, the graphs are typically large and continually evolving. For instance, Facebook has more than 1.3 billion users and approximately 5 new users join Facebook every second [54]; Twitter has more than 300 million users and 3 new users join Twitter every second [54]. Therefore, high efficiency is another requirement for a practical dynamic graph coloring algorithm.

Online graph coloring, which focuses on vertex insertion with the assumption that the color of a vertex is not allowed to change after the color assignment, has been investigated [44] in the literature. However, edge insertion/deletion are common in real applications [2, 19, 36] and vertex insertion can be treated as a special case of edge insertion. Thus, we focus on the general dynamic model in which the graph is updated by edge insertion/deletion. In the literature, an algorithm denoted as DC-Local [60] is proposed for the dynamic graph coloring problem. Briefly, after an edge (u, v) is inserted/deleted, DC-Local locally updates the graph coloring by adjusting only the colors of u and v and their neighbors. The time complexity of DC-Local for each update is $O(\mathsf{dmax}^2)$, where dmax is the maximum vertex degree in the graph. This type of local update strategy may be efficient in practice, but if a new color is introduced in a certain update, the algorithm will miss the opportunity to reduce the number of colors globally and, thus, may continue to increase the number of colors in subsequent updates. Moreover, the graph coloring generated by DC-Local is largely dependent on the order of the edges being inserted/deleted, and may lead to inconsistent coloring if we obtain the same graph by different edge insertion/deletion orders. The example below shows its drawbacks.

Example 1.1: We extract part of a rating network from the *Moive-Lens* dataset (https://movielens.org/). Initially, we color the network using one of the best static graph coloring algorithms for large graphs. The coloring result, with 6 colors, is shown in Fig. 1 (a). Then, as a test, we randomly remove some edges from the graph and add them back in a random order, and repeat this 100 times. Obviously, the final graph is the same as the initial graph. We conduct this test twice using DC-Local to update the graph coloring with the same initial coloring in Fig. 1 (a). The results for the two tests are shown in Fig. 1 (b) and Fig. 1 (c), respectively. We can see that (1) the number of colors is significantly increased in both tests; and (2) the colorings of the two tests are largely different. □

This example shows two main drawbacks of the existing solution: (1) low coloring quality; and (2) inconsistent coloring result. Motivated by this, we aim to design an efficient incremental graph coloring update algorithm that can overcome these drawbacks.

General Idea. Our general idea is simple: after each update of the graph, we aim to update the graph coloring incrementally to make it exactly the same as the coloring result obtained by one of the best static graph coloring algorithms for large graphs, Global [68]. Briefly, given a graph G, Global colors the vertices according to a global vertex order in which vertices are sorted in decreasing order of their degrees in G. For each vertex, Global assigns the vertex the minimum possible color not assigned to its neighbors. Global has been widely adopted in the literature because of its high efficiency in handling large graphs and its high graph coloring quality in practice [1, 3, 55, 70]. To show that our idea is practically applicable, we study two issues: effectiveness and efficiency.

Effectiveness. Our approach is able to overcome the two main drawbacks of the existing algorithm:

- *High Coloring Quality.* Unlike DC-Local, which locally updates vertex colors without considering global optimization, the coloring quality of our approach is the same as Global, which colors the graph in a global vertex order. Therefore, we are able to achieve a much better coloring quality than DC-Local.
- *Consistent Coloring Result.* Given Global's unique global vertex order, its coloring result is only dependent on the graph's topology. Since the coloring result of our approach is the same as Global's, we can guarantee that the coloring result of our approach is independent of the edge deletion/insertion order.

Example 1.2: We conduct the same test in Example 1.1 using our approach on the graph shown in Fig. 1 (a). The initial coloring is computed using Global and the result in Fig. 1 (d) is the same as the coloring in Fig. 1 (a) since the graph topology does not change. □

Efficiency. We design an algorithm that maintains the graph coloring incrementally for each graph update without computing the coloring from scratch using Global. The rationale is based on the observation that, in practice, very few vertices have color changes after an edge insertion/deletion in the new coloring generated by Global. To demonstrate this, we compute the average number of vertices φ with color changes in each update on 10 real datasets from different application domains. According to the results, the maximum φ across the 10 datasets is 40.43 and the average φ is only 11.4 (see Exp-6 in Section 6). This suggests the opportunity to explore only a small number of vertices in the graph to update the graph coloring for each update.

When an edge (u, v) is inserted/deleted, let Δ be union of u, v and the vertices with color changes after the graph update. According to the above discussion, $|\Delta|$ is small in practice. Thus, we aim to explore only those vertices related to Δ to achieve high efficiency. We first propose a color-propagation based algorithm that iteratively recolors a vertex u and notifies its out-neighbors in an oriented coloring graph to be further recolored if the color of u changes. Here, the oriented coloring graph is a directed graph created based on the original graph. By carefully assigning a priority for vertices to be recolored, we can guarantee that each vertex is recolored once, at most, in each update. Such an approach may visit the 2-hop neighbors of vertices in Δ. Thus, we further propose an index that maintains a summary of the color information of the in-neighbors for each vertex in the oriented coloring graph. The index has a linear size to G and can be maintained efficiently. With this index, we can determine whether the color of a vertex will change in constant time prior to the color computation. Thus, the algorithm only needs to explore the vertices in Δ and their neighbors to handle a graph update. The time complexity of our algorithm is $O(n_\Delta \cdot \log(n_\Delta))$ where n_Δ is the number of vertices in Δ and their neighbors.

Contributions. We make the following contributions in this paper.

(1) A new idea to update graph coloring by considering global optimization. We investigate the drawbacks of the existing algorithm using a local update and propose a new idea to update the graph coloring by considering global optimization.

(2) An efficient coloring update algorithm with a bounded time complexity. We propose a color-propagation based algorithm on an auxiliary graph called oriented coloring graph. With a proper vertex propagation order, we bound the explored vertex to be within the 2-hop neighbors of the vertices in Δ.

(3) Novel early pruning strategies to further improve the efficiency. We propose an index, DINC-Index, to efficiently determine whether the color of a vertex will change before color computation occurs and, thus, bound the explored vertices within the neighbors of vertices in Δ. We also explore some pruning rules to reduce the number of propagated vertices to further improve efficiency.

(4) Extensive performance studies on real and synthetic datasets. We conduct extensive performance studies on real and synthetic datasets. The experimental results demonstrate that our proposed algorithm can achieve both high effectiveness and high efficiency.

2. PRELIMINARIES

Consider an undirected and unweighted graph $G = (V, E)$, where $V(G)$ represents the set of vertices and $E(G)$ represents the set of edges in G. We denote the number of vertices as n and the number of edges as m. Every vertex has a unique ID and we use $\mathsf{id}(u, G)$ to denote the id of vertex u. We use $\mathsf{nbr}(u, G)$ to denote the set of neighbors of u for each vertex $u \in V(G)$, i.e., $\mathsf{nbr}(u, G) = \{v|(u, v) \in E(G)\}$. The degree of a vertex $u \in V(G)$, denoted by $\mathsf{deg}(u, G)$, is the number of neighbors of u in G, i.e., $\mathsf{deg}(u, G) = |\mathsf{nbr}(u, G)|$. For simplicity, we omit G in the notations if the context is self-evident. Given a graph G, we use dmax to denote the largest degree of vertices in G. We use $\mathbb{N}$ to denote the set of non-negative integers.

Definition 2.1: (Graph Coloring) Given a graph $G = (V, E)$, a graph coloring of G is a function $f : V \to C$ from the set V of vertices to a set C of colors such that any two incident vertices are assigned different colors, where $C \subset \mathbb{N}$. □

For a graph G and a coloring f, we use $|f(G)|$ to denote the number of colors used in f. For a vertex $u \in V(G)$, we use $u.\mathsf{color} = f(u)$ to denote the color of u assigned by f.

Definition 2.2: (k-colorable) A graph G is k-colorable if there is a graph coloring of G with at most k colors. □

Definition 2.3: (Chromatic Number) For a given graph, the chromatic number of G, denoted by $\chi(G)$, is the smallest integer k for which G is k-colorable. □

Definition 2.4: (Optimal Graph Coloring) For a given graph G, the optimal graph coloring, denoted by $\varrho(G)$, is a graph coloring of G such that $|\varrho(G)| = \chi(G)$. □

Algorithm 1 DC-Local(Graph G)

```
1: Procedure DC-Local-Ins(Graph $G$, Edge$(u, v)$)
2: $G$.insert($(u, v)$);
3: if $u$.color = $v$.color then
4:     $u \leftarrow \mathrm{argmin}_{u' \in \{u,v\}}\{|\mathsf{SC}(u')|\}$;
5:     DC-Local-Recolor($u$);
6: Procedure DC-Local-Del(Graph $G$, Edge$(u, v)$)
7: $G$.delete($(u, v)$);
8: DC-Local-Recolor($u$); DC-Local-Recolor($v$);
9: Procedure DC-Local-Recolor(Vertex $u$)
10: $\mathbb{C} \leftarrow \mathsf{SC}(u)$; $\mathbf{C} \leftarrow \{0, 1, \dots, \mathsf{deg}(u)\}$;
11: $c_{\max} \leftarrow \max\{c|c \in \mathbb{C}\}$; $c_{\min} \leftarrow \min\{c|c \in \mathbf{C}, c \notin \mathbb{C}\}$;
12: if $c_{\min} < c_{\max}$ then
13:     $u$.color $\leftarrow c_{\min}$;
14: else
15:     mcolor[$c$] $\leftarrow 0$ for all $c \in \mathbb{C}$;
16:     for all $v \in \mathsf{nbr}(u)$ do
17:         mcolor[$v$.color] $\leftarrow \max\{$mcolor[$v$.color], $|\mathsf{SC}(v)|\}$;
18:     $c_{\mathsf{cand}} \leftarrow \mathrm{argmin}_{c \in \mathbb{C}}\{$mcolor[$c$]$\}$;
19:     if $c_{\mathsf{cand}} < c_{\min} - 1$ then
20:         $u$.color $\leftarrow c_{\mathsf{cand}}$;
21:         for each $v \in \mathsf{nbr}(u)$ if $v$.color = $c_{\mathsf{cand}}$ then
22:             $v$.color $\leftarrow$ SmallestUnassignedColor($v$);
23:     else
24:         $u$.color $\leftarrow c_{\min}$;
25: Procedure SmallestUnassignedColor(Graph $G$, Vertex $u$)
26: $\mathbf{C} \leftarrow \{0, 1, \dots, \mathsf{deg}(u)\}$, $\mathbb{C} \leftarrow \emptyset$;
27: for each $v \in \mathsf{nbr}(u)$ do
28:     $\mathbb{C} \leftarrow \mathbb{C} \cup \{v$.color$\}$;
29: return $\min\{c|c \in \mathbf{C}, c \notin \mathbb{C}\}$;
```

Problem Statement. In this paper, we study the problem of dynamic graph coloring, which is defined as follows: Given a graph G, compute the optimal graph coloring $\varrho(G)$ of G when G is dynamically updated by the insertion and deletion of edges.

Since computing the optimal graph coloring is an NP-hard problem [27], we resort to approximate solutions in this paper.

3. THE EXISTING SOLUTION

The state-of-the-art dynamic graph coloring algorithm is proposed in [60], which is based on saturation colors of a vertex.

Definition 3.1: (Saturation Colors) Given a graph G and a graph coloring f, for a vertex $u \in V(G)$, the saturation colors of u, denoted by $\mathsf{SC}(u)$, is the set of colors that f assigns to u's neighbors, i.e., $\mathsf{SC}(u) = \bigcup_{v \in \mathsf{nbr}(u)} v.\mathsf{color}$. □

DC-Local is shown in Algorithm 1. When an edge (u, v) is inserted, if u and v share the same color, DC-Local recolors the vertex with the small number of saturation colors (line 1-5). When an edge (u, v) is deleted, DC-Local recolors both u and v (line 6-8).

To recolor a specific vertex u, DC-Local tries to avoid increasing the number of colors based on $\mathsf{SC}(u)$. Specifically, it first computes the smallest color $c_{\min}$ that is not assigned to any neighbor of u (line 10-11). If $c_{\min}$ is smaller than the maximum color in $\mathsf{SC}(u)$, $c_{\min}$ is assigned to u (line 12-13). Otherwise, for each u's neighbor v, DC-Local uses mcolor[v.color] to store the the maximal number of the saturation colors of u's neighbors that are assigned with v.color (line 15-17). DC-Local finds the color c_{cand} that is assigned to a neighbor v of u and the number of $\mathsf{SC}(v)$ is smaller than that of any other neighbors of u which is not assigned with color c_{cand} based on mcolor (line 18). If c_{cand} is smaller than $c_{\min}$-1, DC-Local assigns c_{cand} to u and all the neighbors of u whose color is c_{cand} are reassigned with the smallest color not assigned to their neighbors (line 19-22). Otherwise, it colors u with $c_{\min}$ (line 24). For a given vertex u, procedure SmallestUnassignedColor computes the smallest color not assigned to any neighbor of u (line 25-29).

Theorem 3.1: *The time complexity of* DC-Local *to handle an edge insertion/deletion is* $O(\mathsf{dmax}^2)$. □

Proof: Let's consider the edge insertion first. For a vertex u, $\mathsf{SC}(u)$ and $\mathsf{SmallestUnassignedColor}(u)$ can be computed in $O(\mathsf{dmax})$. Thus, $\mathsf{DC\text{-}Local\text{-}Recolor}(u)$ can finish in $O(\mathsf{dmax}^2)$. Then, the time complexity of DC-Local-Ins for an edge insertion is $O(\mathsf{dmax}^2)$. The edge deletion can be proved similarly. Thus, the theorem holds. □

Drawbacks of DC-Local. DC-Local maintains the graph coloring by only considering recoloring the neighbors of the vertices in the inserted/deleted edge. However, it has two drawbacks:

(D_1) Inferior Coloring Quality. The assumption behind DC-Local is that the graph coloring can be well approximated by only exploring local neighborhood of the vertices in the inserted/deleted edge. However, as shown in Fig. 1, this assumption does not generally hold in practice since it may miss opportunities to globally reduce the number of colors.

(D_2) Coloring Inconsistency. As shown in Fig. 1, the graph coloring generated by DC-Local cannot keep consistent if we obtain the same graph with different edge insertion/deletion orders. This means the graph coloring generated by DC-Local is not robust in practice.

4. A NEW APPROACH

To overcome the drawbacks of DC-Local discussed in Section 3, we devise a new algorithm for dynamic graph coloring. In this section, we first analyze the dynamic graph coloring problem and propose a basic algorithm. Then, we further improve the basic algorithm with a prioritized vertex exploration.

4.1 The General Idea

The key idea of our approach is that we aim to guarantee the quality by making the coloring result consistent with one of the best static graph coloring algorithms for large graphs. Since the optimal graph coloring problem is an NP-hard problem [27], and there is no polynomial-time $n^{1-\epsilon}$ approximation algorithm for the optimal graph coloring problem, unless NP=ZPP [72], existing static graph coloring algorithms resort to the greedy approach. One of the best algorithms for large graphs is Global [68]. It is widely used in the literature because of its high efficiency and good coloring quality in practice [55, 70, 1, 3], which is also verified in our experiments. Therefore, we use it to design our approach. Global works as follows: It first sorts the vertices in decreasing order of their degrees (increasing order of their IDs for vertices with the same degree). Then, it iterates vertices in the sorted order and assigns each vertex the minimum possible color not assigned to its already colored neighbors. The essence of Global is to find a coloring f such that the color of vertex satisfies the following property:

Definition 4.1: (Global Color Property γ) Given a graph G and a coloring f, the color of u satisfies the global color property γ of G, denoted by $u.\mathsf{color} \models \gamma(G)$, if $u.\mathsf{color} = \min\{c \mid c \in \mathbb{N}, c \notin C(u)\}$, where $C(u) = \{v.\mathsf{color} \mid v \in \mathsf{nbr}(u) \wedge (\mathsf{deg}(v) > \mathsf{deg}(u) \vee (\mathsf{deg}(v) = \mathsf{deg}(u) \wedge \mathsf{id}(v) < \mathsf{id}(u)))\}$. □

The dynamic graph coloring problem can be redefined as:

Definition 4.2: (Problem Definition*) Given a graph G, we aim to maintain a graph coloring f such that for each vertex $u \in V(G)$, $u.\mathsf{color} \models \gamma(G)$ when G is dynamically updated by the insertion and deletion of edges. □

The approach designed based on Definition 4.2 can achieve the goal of coloring quality and coloring consistency as the coloring satisfying Definition 4.2 is the same as the coloring generated by Global. A naive approach to maintain such a coloring is to use Global to recompute the graph coloring for every update. Obviously, such an approach is impractical for large graphs. Therefore, an incremental algorithm that maintains the global color property is needed. A straightforward solution is to identify the set of vertices that violates the global color property and then recolor them. However, recoloring a certain vertex may trigger other violations of the global color property. To efficiently identify the vertex recoloring order, we introduce the *oriented coloring graph*.

4.2 Oriented Dynamic Graph Coloring

Oriented Coloring Graph. Oriented coloring graph is constructed based on the total order of vertices defined as follows:

Definition 4.3: (Total Order $\prec$) Given a graph G and two vertices $u, v \in G$, we define $u \prec v$ if
- $\mathsf{deg}(u) > \mathsf{deg}(v)$, or
- $\mathsf{deg}(u) = \mathsf{deg}(v)$ and $\mathsf{id}(u) < \mathsf{id}(v)$.

Obviously, $\prec$ defines a total order of all vertices in G. □

For two vertices u and v, if $u \prec v$, we say u dominates v and v is dominated by u. Based on Definition 4.3, we can assign a direction to each edge in G with respect to the total order $\prec$ and we define:

Definition 4.4: (Oriented Coloring Graph) Given a graph $G = (V, E)$, the Oriented Coloring Graph (OCG) $G^* = (V, E^*)$ of G is a directed acyclic graph such that for each edge $(u, v) \in E$, if $u \prec v$ ($v \prec u$), there is a directed edge from u to v (from v to u) in G^*, denoted by $<u, v>$ ($<v, u>$). □

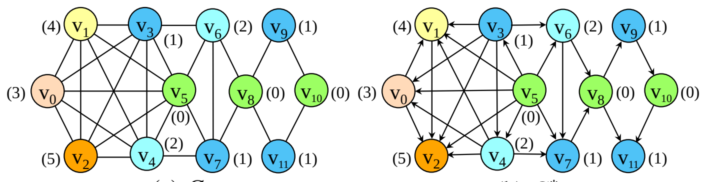
If there is a directed edge $<u, v>$ in G^*, we say u is an in-neighbor of v and v is an out-neighbor of u. For each vertex $u \in G^*$, we use $\mathsf{nbr}^-(u, G^*)$ and $\mathsf{nbr}^+(u, G^*)$ to denote the set of its in-neighbors and out-neighbors in G^* respectively. And we use $\mathsf{nbr}(u, G^*)$ to denote $\mathsf{nbr}^-(u, G^*) \cup \mathsf{nbr}^+(u, G^*)$. For a vertex u, the in-degree of u, denoted by $\mathsf{deg}^-(u, G^*)$, is the number of in-neighbors of u and the out-degree of u, denoted by $\mathsf{deg}^+(u, G^*)$, is the number of out-neighbors of u. For simplicity, we omit G^* from the notations if the context is self-evident. When an edge $<u, v>$ is inserted into/deleted from G^*, we use $G^*+<u, v>/G^*-<u, v>$ to represent the new OCG after the update. Based on the total order $\prec$ used to define G^*, we can easily derive that G^* is a directed acyclic graph (DAG).

Definition 4.5: (OCG **Coloring)** Given an OCG $G^* = (V, E^*)$, an OCG coloring is a coloring f in which any two incident vertices $u, v \in V$ are assigned with different colors, i.e., $<u, v> \in E^* \Rightarrow u.\mathsf{color} \neq v.\mathsf{color}$. □

Based on Definition 4.5, it is obvious that f is an OCG coloring of G^* if and only if f is a graph coloring of G. And we also define the oriented global color property on OCG:

Definition 4.6: (Oriented Global Color Property σ) Given an OCG G^* and a coloring f, the color of u satisfies oriented global color property σ of G^*, denoted by $u.\mathsf{color} \models \sigma(G^*)$, if $u.\mathsf{color} = \min\{c | c \in \mathbb{N}, c \notin \bigcup_{v \in \mathsf{nbr}^-(u)} v.\mathsf{color}\}$. □

Based on Definition 4.6, our problem (Definition 4.2) is equivalent to maintaining the oriented global color property for all vertices in the OCG. For simplicity, we call the OCG coloring f of G^* in which $u.\mathsf{color} \models \sigma(G^*)$ for all $u \in V(G^*)$ as the *global oriented coloring* of G^* and denote it by $\Sigma(G^*)$. Our aim is to maintain the global oriented coloring $\Sigma(G^*)$ when G^* is dynamically updated.



(a) G (b) G^*

Figure 2: Oriented Coloring Graph

Example 4.1: Consider G in Fig. 2 (a), the OCG G^* of G is shown in Fig. 2 (b). In G^*, the direction of an edge is decided by $\prec$. For example, as $v_5 \prec v_0$, we create a directed edge $<v_5,v_0>$ in G^*. We also show $\Sigma(G^*)$ in Fig. 2 (b). The color of each vertex is shown in the parentheses near the vertex. It is clear that the color of each vertex in $\Sigma(G^*)$ also satisfies the global color property of G. □

Given the OCG G^*, when an edge $<u,v>$ is inserted/deleted, we aim to compute $\Sigma(G^* \pm <u,v>)$ by recoloring the vertices whose colors in $\Sigma(G^*)$ violate $\sigma(G^* \pm <u,v>)$. According to Definition 4.6, it is obvious that the colors of the vertices that always dominate u before and after the update remain the same in $\Sigma(G^*)$ and $\Sigma(G^* \pm <u,v>)$. Thus, these vertices do not need to be recolored. However, the colors of other vertices in $\Sigma(G^*)$ may violate $\sigma(G^* \pm <u,v>)$. To maintain the oriented global color property, we recolor the vertices in $G^* \pm <u,v>$ using the following equation:

$$f_{\mathsf{new}}(w) \leftarrow \min\{c \mid c \in \mathbb{N}, c \notin \cup_{x \in \mathsf{nbr}^-(w)} f_{\mathsf{old}}(x)\} \quad (1)$$

where f_{new} and f_{old} are the graph colorings before and after the recoloring of w respectively. Here, for brevity, although it is possible that two incident vertices have the same color in f_{new} (f_{old}), we still refer to f_{new} (f_{old}) as a graph coloring. Based on Eq. 1, we have:

Lemma 4.1: *For a given G^* and $\Sigma(G^*)$, when an edge $<u,v>$ is inserted/deleted, the coloring f when Eq. 1 converges for all vertices $w \in G^*$ is $\Sigma(G^* \pm <u,v>)$.* □

Proof: We prove this by contradiction. If the coloring is not $\Sigma(G^* \pm <u,v>)$, which means there exists a vertex whose color violates $\sigma(G^* \pm <u,v>)$. This is contradicts with the given condition that Eq. 1 converges. Thus, the lemma holds. □

According to Lemma 4.1, we can obtain $\Sigma(G^* \pm <u,v>)$ by iteratively recoloring the vertices whose colors violate $\sigma(G^* \pm <u,v>)$. The remaining problem is how to do this efficiently. Below, we introduce a color propagation mechanism on OCG G^*.

Color Propagation by the CAN Step. According to Eq. 1, a vertex w needs to be recolored only if one of its in-neighbors changes its color. Therefore, when a vertex w changes its color, we only need to notify its out-neighbors as the *candidates* to be recolored. We do this using a CAN step with three operators CC, AC, and NC.

Definition 4.7: (Operator CC) Given an OCG G^* and a vertex u in G^*, the CC operator Collects the Colors $\mathbb{C}$ of u's in-neighbors, i.e., it computes $\mathbb{C} = \bigcup_{v \in \mathsf{nbr}^-(u)} v.\mathsf{color}$. □

Definition 4.8: (Operator AC) Given an OCG G^*, a vertex u in G^*, and a set of colors $\mathbb{C}$, the AC operator Assigns the Color of u to be the smallest color not in $\mathbb{C}$. It returns true if the color of u changes and returns false otherwise. □

Definition 4.9: (Operator NC) Given an OCG G^*, a vertex u in G^*, and a boolean indicator b, the NC operator Notifies the out-neighbors of u to reassign their Colors if b is true. □

Definition 4.10: (A CAN Step) Given an OCG G^* and a vertex u in G^*, a CAN step performs CC, AC and NC on u sequentially. □

Since G^* is a DAG, we can guarantee that *the color propagation using the* CAN *steps will not result in propagation loops.*

The Seed Vertices Selection. To start the color propagation using the CAN step, we first need to select a set of seed vertices. It is worth noting that when an edge $<u,v>$ is inserted/deleted, it is not enough to just consider u and v as the seed vertices. This is because after $<u,v>$ is inserted/deleted, the degree of u and v will change. As a result, the dominant relationship between u (v) and their neighbors will change. Consequently, these vertices whose dominant relationship with respect to u (v) are changed may also violate $\sigma(G^* \pm <u,v>)$ and, thus, need to be considered as the seed vertices as well. Specifically, we use the following two lemmas to select the seed vertices for an edge insertion/deletion respectively:

Lemma 4.2: *Given an* OCG G^**, after inserting an edge $<u,v>$, it is adequate to consider $\{\{u,v\} \cup I_u \cup I_v\}$ as the seed vertices to compute $\Sigma(G^* + <u,v>)$, where $I_u = \mathsf{nbr}^-(u, G^*) \cap \mathsf{nbr}^+(u, G^* + <u,v>)$ and $I_v = \mathsf{nbr}^-(v, G^*) \cap \mathsf{nbr}^+(v, G^* + <u,v>)$.* □

Proof: We can prove this by contradiction. Suppose that it is inadequate to consider $\{\{u,v\} \cup I_u \cup I_v\}$ as seed vertices to compute $\Sigma(G^* + <u,v>)$, which means there exists a vertex $w \notin \{\{u,v\} \cup I_u \cup I_v\}$ and its color in $\Sigma(G^*)$ and $\Sigma(G^* + <u,v>)$ are different, but it is not notified by a CAN step during the color propagation procedure. According to Eq. 1, the vertices in $\{\{u,v\} \cup I_u \cup I_v\}$ lead to the color propagation as their in-neighbor are changed in $G^* + <u,v>$. Based on the definition of CAN step, a vertex is not notified iff the colors of its in-neighbors are not changed during the propagation. As w is not notified during the color propagation procedure, we can derive that for all the in-neighbors of w, their colors are the same in $\Sigma(G^*)$ and $\Sigma(G^* + <u,v>)$. Then we can derive that the colors of w's in-neighbors in $\Sigma(G^*)$ and $\Sigma(G^* + <u,v>)$ are the same but the color of w in $\Sigma(G^*)$ and $\Sigma(G^* + <u,v>)$ are different, which contradicts with Definition 4.6. Thus, the lemma holds. □

Lemma 4.3: *Given an* OCG G^**, after deleting an edge $<u,v>$, it is adequate to consider $\{\{u,v\} \cup D_u \cup D_v\}$ as the seed vertices to compute $\Sigma(G^* - <u,v>)$, where $D_u = \mathsf{nbr}^+(u, G^*) \cap \mathsf{nbr}^-(u, G^* - <u,v>)$ and $D_v = \mathsf{nbr}^+(v, G^*) \cap \mathsf{nbr}^-(v, G^* - <u,v>)$.* □

Proof: This lemma can be proved similarly as Lemma 4.2. □

Algorithm Design. Our DC-Orient algorithm to maintain $\Sigma(G^*)$ is shown in Algorithm 2. It contains two main procedures, DC-Orient-Ins and DC-Orient-Del, to handle the edge insertion and deletion respectively. Both DC-Orient-Ins and DC-Orient-Del maintain a queue q to store the candidate vertices that need to be recolored (line 2/line 8). When an edge $<u,v>$ is inserted/deleted, DC-Orient-Ins/DC-Orient-Del first invokes OCG-Ins/OCG-Del (introduced in Algorithm 3) to maintain the OCG G^* and obtain the seed vertices in Lemma 4.2/Lemma 4.3 (line 3/line 9). It pushes these vertices into q (line 4-5/line 10-11), and then invokes the CAN procedure to iteratively recolor the vertices and conduct color propagation using the CAN step (line 6/line 12).

The CAN procedure iteratively processes the CAN step (line 16-18) and maintains the candidate vertices to be recolored in q. The recoloring procedure terminates when there is no vertex in q (line 14). In a certain CAN step, the CC operator (line 16), the AC operator (line 17), and the NC operator (line 18) are performed sequentially.

The CC operator is implemented as procedure CollectColor(u) (line 19-23). It simply collects the set of colors $\mathbb{C}$ from the in-neighbors of u and returns $\mathbb{C}$ as defined in Definition 4.7. The AC operator is implemented as the AssignColor(u, $\mathbb{C}$) procedure (line 24-29). According to Definition 4.8, it first computes the smallest color c_{new} that is not in $\mathbb{C}$ (line 25-26). If $c_{\mathsf{new}} \neq u.\mathsf{color}$, it assigns c_{new} to u and returns true; otherwise it returns false (line 27-29). The NC

Algorithm 2 DC-Orient(OCG G^*)

```
1:  Procedure DC-Orient-Ins(OCG $G^*$, Edge<$u$, $v$>)
2:    Queue $q \leftarrow \emptyset$;
3:    S ← OCG-Ins($G^*$,<$u$, $v$>) (Algorithm 3);
4:    for each $w \in$ S do
5:      $q$.push($w$);
6:    CAN ($G^*$, $q$);
7:  Procedure DC-Orient-Del(OCG $G^*$, Edge<$u$, $v$>)
8:    Queue $q \leftarrow \emptyset$;
9:    S ← OCG-Del($G^*$,<$u$, $v$>) (Algorithm 3);
10:   for each $w \in$ S do
11:     $q$.push($w$);
12:   CAN($G^*$, $q$);
13: Procedure CAN(OCG $G^*$, Queue $q$)
14:   while $q \neq \emptyset$ do
15:     $u \leftarrow q$.pop();
16:     $\mathbb{C} \leftarrow$ CollectColor($u$);         //line 16-18 is a CAN step for $u$
17:     $b \leftarrow$ AssignColor($u$, $\mathbb{C}$);
18:     NotifyColor($u$, $b$, $q$);
19: Procedure CollectColor(Vertex $u$)                     //the CC operator
20:   $\mathbb{C} \leftarrow \emptyset$;
21:   for each $v \in$ nbr$^-(u)$ do
22:     $\mathbb{C} \leftarrow \mathbb{C} \cup \{v.\text{color}\}$;
23:   return $\mathbb{C}$;
24: Procedure AssignColor(Vertex $u$, Set $\mathbb{C}$)               //the AC operator
25:   C ← $\{0, 1, \ldots, \text{deg}(u)\}$;
26:   $c_{\text{new}} \leftarrow \min\{c | c \in \mathbf{C}, c \notin \mathbb{C}\}$;
27:   if ($c_{\text{new}} \neq u$.color)
28:     $u$.color $\leftarrow c_{\text{new}}$; return true;
29:   else return false;
30: Procedure NotifyColor(Vertex $u$, Bool $b$, Queue $q$)  //the NC operator
31:   if ($b$ = true) then
32:     for each $v \in$ nbr$^+(u)$ do
33:       if $v \notin q$ then $q$.push($v$);
```

operator is implemented as procedure NotifyColor(u, b, q). Here b indicates whether the color of vertex u changes, and q is the queue. According to Definition 4.9, if the color of u changes, the procedure notifies all the out-neighbors of u to recolor by pushing them into q if they have not been in q (line 31-33).

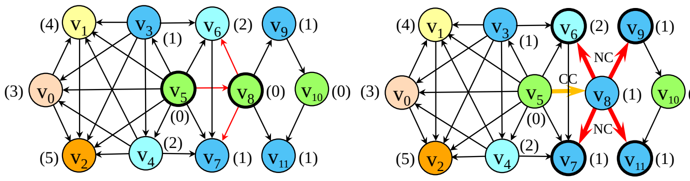
OCG **Maintenance.** OCG-Maintain (Algorithm 3) maintains the OCG G^* and returns the vertices defined in Lemma 4.2/Lemma 4.3. It contains two procedures, OCG-Ins and OCG-Del, to handle the edge insertion and deletion respectively.

OCG-Ins uses **S** to store the vertices in Lemma 4.2. When an edge <u, v> is inserted, OCG-Ins stores u and v in **S** based on Lemma 4.2 (line 2), and inserts edge <u, v> into G^* (line 3). As the degree of vertices u and v increases by 1, the direction of edges involving u or v may change based on Definition 4.3. OCG-Ins adjusts the direction of the edges involving u or v in line 4-8. Take the vertex u as an example. Since the degree of u increases, it is possible that the vertices that belong to nbr$^-(u)$ before inserting <u, v> belong to nbr$^+(u)$ after the insertion. OCG-Ins visits each vertex $u' \in$ nbr$^-(u)$ to check the dominant relationship between u and u'. If their dominant relationship changes after the edge insertion (line 5), OCG-Ins adjusts the direction of the edge (line 6) and adds u' in **S** (line 7). Finally, OCG-Ins returns **S** in line 9. OCG-Del handles edge deletion similarly as OCG-Ins (line 10-18).

Example 4.2: Recall the OCG G^* in Fig. 2 (b) and suppose that an edge <v_5, v_8> is inserted. OCG-Ins first inserts <v_5, v_8> into G^*. After inserting <v_5, v_8>, the degree of v_8 increases from 4 to 5. As a result, the dominant relationship between v_6 and v_8 is changed. Therefore, OCG-Ins changes <v_6, v_8> to <v_8, v_6> in Fig. 3 (a). The changed edges are shown as red lines in Fig. 3 (a). OCG-Ins returns the set $\{v_5, v_8, v_6, v_7\}$ based on Lemma 4.2.

Algorithm 3 OCG-Maintain(OCG G^*)

```
1:  Procedure OCG-Ins(OCG $G^*$, Edge<$u$, $v$>)
2:    S ← $\emptyset$; S ← S ∪ $u$; S ← S ∪ $v$;
3:    add edge <$u$, $v$> in $G^*$;
4:    for each $u' \in$ nbr$^-(u)$ do
5:      if $u \prec u'$ then
6:        remove edge <$u'$, $u$> and add edge <$u$, $u'$> in $G^*$;
7:        S ← S ∪ $u'$;
8:    process line 4-7 by replacing $u$ with $v$ and $u'$ with $v'$;
9:    return S;
10: Procedure OCG-Del(OCG $G^*$, Edge<$u$, $v$>)
11:   S ← $\emptyset$; S ← S ∪ $u$; S ← S ∪ $v$;
12:   remove edge <$u$, $v$> in $G^*$;
13:   for each $u' \in$ nbr$^+(u)$ do
14:     if $u' \prec u$ then
15:       remove edge <$u$, $u'$> and add edge <$u'$, $u$> in $G^*$;
16:       S ← S ∪ $u'$;
17:   process line 13-16 by replacing $u$ with $v$ and $u'$ with $v'$;
18:   return S;
```



(a) OCG maintenance (b) A CAN step on v_8

Figure 3: Insertion of edge <v_5, v_8>

Step	color												q
	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	
Init	3	4	5	1	2	0	2	1	0	1	0	1	v_5, v_8, v_6, v_7
1.CAN(v_5)	3	4	5	1	2	0	2	1	0	1	0	1	v_8, v_6, v_7
2.CAN(v_8)	3	4	5	1	2	0	2	1	1	1	0	1	v_6, v_7, v_9, v_{11}
3.CAN(v_6)	3	4	5	1	2	0	2	1	1	1	0	1	v_7, v_9, v_{11}
4.CAN(v_7)	3	4	5	1	2	0	2	3	1	1	0	1	v_9, v_{11}
5.CAN(v_9)	3	4	5	1	2	0	2	3	1	0	0	1	v_{11}, v_{10}
6.CAN(v_{11})	3	4	5	1	2	0	2	3	1	0	0	2	v_{10}
7.CAN(v_{10})	3	4	5	1	2	0	2	3	1	0	1	1	v_{11}
8.CAN(v_{11})	3	4	5	1	2	0	2	3	1	0	1	0	$\emptyset$

Figure 4: Steps of DC-Orient **for inserting edge <v_5, v_8>**

Fig. 3 (b) shows a CAN step on vertex v_8. It first collects the color set $\mathbb{C}$ of v_8's in-neighbors using the CC operator. v_5 is the only in-neighbors of v_8 and its color is 0, thus $\mathbb{C} = \{0\}$. As the smallest color not in $\mathbb{C}$ is 1, AC changes the color of v_8 from 0 to 1 and returns the true indicator for NC. NC notifies the set of out-neighbors v_6, v_7, v_9, v_{11} of v_8 by pushing them into q.

The recoloring procedure of DC-Orient-Ins is shown in Fig. 4. The vertex that the CAN step processes and the color of each vertex is shown for each step, along with the vertices in q afterwards. For example, at step 2, after the CAN step for v_8, the color of v_8 is changed from 0 to 1 and two new vertices v_9 and v_{11} are pushed into q. DC-Orient-Ins finishes the recoloring in 8 steps. □

Theorem 4.1: *For a given* OCG G^* *and* $\Sigma(G^*)$*, when an edge* <u, v> *is inserted/deleted, the coloring returned by Algorithm 2 is* $\Sigma(G^* \pm$<u, v>$)$. □

Proof: When an edge <u, v> is inserted, OCG-Ins can correctly maintain G^* and return the vertices based on Lemma 4.2 (line 3). Line 16-18 implements a CAN step. As a CAN step recolors a vertex based on Eq. 1 and we iteratively process the vertices whose colors may violate $\sigma(G^*+$<u, v>$)$ until there exists no such kind of vertices. According to Lemma 4.1, the coloring is $\Sigma(G^*+$<u, v>$)$ when the recoloring procedure converges. Thus, DC-Orient-Ins returns $\Sigma(G^*+$<u, v>$)$. Edge deletion can be proved similarly. □

Since $\Sigma(G^*)$ is only dependent on the topology of graph G^*, it is easy to see that Algorithm 2 can guarantee coloring consistency.

Theorem 4.2: *Let n_o be the number of vertices pushed in q in Algorithm 2, the time complexity of Algorithm 2 to handle the insertion/deletion of an edge $<u, v>$ is $O(n_o \cdot \mathsf{dmax})$.* □

Proof: Let's consider the edge insertion first. For an inserted edge, DC-Orient-Ins first invokes OCG-Ins to maintain G^* (line 3). Since OCG-Ins only visits the in-neighbors of u and v, it can be finished in $O(\mathsf{dmax})$. The push/pop operation for a queue can be finished in $O(1)$. In the recoloring procedure, we push n_o vertices in q (line 4-5 and 33) and pop n_o vertices from q (line 15), which can be finished in $O(n_o)$. For each vertex u in q, both CollectColor (line 16) and AssignColor (line 17) can be finished in $O(\mathsf{dmax})$. Thus, the time for processing n_o vertices is $O(n_o \cdot \mathsf{dmax})$. Thus, the time complex for an edge insertion is $O(n_o \cdot \mathsf{dmax})$. Edge deletion can be proved similarly. Thus, the theorem holds. □

4.3 Prioritized Dynamic Graph Coloring

As shown in Theorem 4.2, the time complexity of Algorithm 2 depends on the number of vertices pushed in q. However, such a number is not bounded since a vertex may be pushed into q multiple times. We call it the out-of-order NC problem.

Out-of-Order NC Problem. For a given OCG G^*, when an edge $<u, v>$ is inserted/deleted, the reason that a vertex w may be pushed into q multiple times in Algorithm 2 is that the colors of multiple in-neighbors of w are changed, which leads to w being pushed into q repeatedly by the NC operator. For instance, in Example 4.2, vertex v_{11} has two in-neighbors v_8 and v_{10}. At CAN step 2, vertex v_8 is recolored and notifies v_{11} to be pushed into q. At CAN step 6, vertex v_{11} is popped out from q. However, at CAN step 7, vertex v_{10} is recolored and notifies v_{11} to be pushed into q again. As a result, v_{11} is pushed into q twice.

Prioritized Dynamic Graph Coloring. From the above discussion, we can see the out-of-order NC problem is caused by the situation in which a vertex is recolored before one of its in-neighbors. In Example 4.2, the vertex v_{11} is recolored at step 6 while its in-neighbors v_{10} is recolored at step 7, which causes v_{11} to be recolored again. To resolve this problem, we need to postpone the recoloring of a vertex until all its candidate in-neighbors have been recolored. In other words, we need to find an appropriate order of vertices to be recolored such that *when recoloring a certain vertex, all its candidate in-neighbors have been recolored*. Note that the OCG G^* is a directed acyclic graph. Therefore, if we follow a topological order of vertices in the directed acyclic graph to recolor the vertices, the above condition can always be satisfied. As a result, the out-of-order NC problem can be completely avoided.

Algorithm Design. We can obtain the topological order by assigning each vertex a priority in the queue q. Since the direction of the edges in G^* are assigned according to the $\prec$ relation, we can simply use the $\prec$ relation to define the vertex priority as follows.

Definition 4.11: (Vertex Priority) Given two vertices u and v, if $u \prec v$, then u has a higher priority than v in q. □

The prioritized dynamic graph coloring algorithm is shown in Algorithm 4. It contains two procedures, DC-Pri-Ins and DC-Pri-Del, to handle an edge insertion and deletion, respectively. DC-Pri-Ins (DC-Pri-Del) shares a similar framework to DC-Orient-Ins (DC-Orient-Del) except that the queue is replaced with a priority queue at line 2 (line 7). Here, the priority of vertices in the priority queue is based on Definition 4.11.

Algorithm 4 DC-Pri(OCG G^*)

```
1: Procedure DC-Pri-Ins(OCG $G^*$, Edge<$u$, $v$>)
2:   PriorityQueue $q \leftarrow \emptyset$;
3:   S ← OCG-Ins($G^*$,<$u$, $v$>); (Algorithm 3)
4:   for each $w \in$ S do $q$.push($w$);
5:   CAN($G^*$, $q$); (Algorithm 2)
6: Procedure DC-Pri-Del(OCG $G^*$, Edge<$u$, $v$>)
7:   PriorityQueue $q \leftarrow \emptyset$;
8:   S ← OCG-Del($G^*$,<$u$, $v$>); (Algorithm 3)
9:   for each $w \in$ S do $q$.push($w$);
10:  CAN($G^*$, $q$); (Algorithm 2)
```

Step	color												q
	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	
Init	3	4	5	1	2	0	2	1	0	1	0	1	v_5, v_8, v_6, v_7
1.CAN(v_5)	3	4	5	1	2	0	2	1	0	1	0	1	v_8, v_6, v_7
2.CAN(v_8)	3	4	5	1	2	0	2	1	1	1	0	1	v_6, v_7, v_9, v_{11}
3.CAN(v_6)	3	4	5	1	2	0	2	1	1	1	0	1	v_7, v_9, v_{11}
4.CAN(v_7)	3	4	5	1	2	0	2	3	1	1	0	1	v_9, v_{11}
5.CAN(v_9)	3	4	5	1	2	0	2	3	1	0	0	1	v_{10}, v_{11}
6.CAN(v_{10})	3	4	5	1	2	0	2	3	1	0	1	1	v_{11}
7.CAN(v_{11})	3	4	5	1	2	0	2	3	1	0	1	0	$\emptyset$

Figure 5: Steps of DC-Pri for inserting edge <v_5, v_8>

Example 4.3: Continuing with OCG G^* in Fig. 2 (b) to demonstrate the process of DC-Pri, suppose that the edge $<v_5, v_8>$ is inserted, the corresponding G^* after the insertion of $<v_5, v_8>$ is the same as that in Example 4.2 shown in Fig. 3 (a). When $<v_5, v_8>$ is inserted, the recoloring procedure of DC-Pri-Ins is shown in Fig. 5. For each step, we show the vertex on which the CAN step process, the color of each vertex, and the vertices in the priority queue q after the CAN step. Because of the vertex priority, the vertex v_{11} is recolored after its in-neighbor v_{10}. Therefore, v_{11} is only recolored once. As a result, DC-Pri-Ins finishes the recoloring procedure in 7 steps. □

Theorem 4.3: *For a given OCG G^*, when an edge $<u, v>$ is inserted/deleted, let Δ be the union of u, v and the vertices whose colors in $\Sigma(G^*)$ and $\Sigma(G^* \pm <u, v>)$ are different, then the number of vertices pushed in q by Algorithm 4 can be bounded by n_Δ where $n_\Delta = |\cup_{u \in \Delta} \mathsf{nbr}(u) \cup \Delta|$* □

Proof: Let's consider the edge insertion first. In Algorithm 4, the number of vertices pushed in q in line 4 can be bounded by $|\cup_{u \in \{u,v\}} \mathsf{nbr}(u) \cup \{u, v\}|$. According to Theorem 4.1, the vertices whose colors are different in $\Sigma(G^*)$ and $\Sigma(G^* \pm <u, v>)$ are pushed into q. Besides, based on Definition 4.10, the neighbors of the vertex whose colors are different are pushed into q by operator NC. Since Algorithm 4 avoids the out-of-order problem, each vertex is processed only once, which means the vertices pushed into q are just the vertices in Δ together with their out-neighbors. Therefore, the number of vertices pushed in q can be bounded by $|\cup_{u \in \Delta} \mathsf{nbr}(u) \cup \Delta|$. Edge deletion can be proved similarly. Thus, the theorem holds. □

Theorem 4.4: *The time complexity of Algorithm 4 to handle the insertion/deletion of an edge $<u, v>$ is $O(n_\Delta \cdot (\mathsf{dmax} + \log(n_\Delta)))$.* □

Proof: Let's consider the edge insertion first. Let η be the number of vertices pushed into q. For an edge insertion, DC-Orient-Ins first invokes OCG-Ins to maintain G^* (line 3), which can be finished in $O(\mathsf{dmax})$. The push/pop operation for a priority queue can be finished in $O(1)/O(\log \eta)$ if we use Fibonacci heap. We push/pop η vertices into/from q, which can be finished in $O(\eta \cdot \log \eta)$. For each vertex u in q, both CollectColor and AssignColor can be finished in $O(\mathsf{dmax})$. Thus, the time for processing η vertices is $O(\eta \cdot \mathsf{dmax})$. Since η can be bounded by n_Δ based on Theorem 4.3, the time complexity of DC-Pri-Ins is $O(n_\Delta \cdot (\mathsf{dmax} + \log(n_\Delta)))$. The edge deletion can be proved similarly. Thus, the theorem holds. □

5. EARLY PRUNING

In this section, we aim to further improve the performance of our algorithm using early pruning strategies. In Theorem 4.4, the time complexity of Algorithm 4 depends on two factors: n_Δ and dmax. Although n_Δ can be well bounded according to Theorem 4.3, dmax can be large. In this section, we try to eliminate the factor dmax from the time complexity and further reduce n_Δ. In DC-Pri, two types of vertices are pushed in the priority queue q:

- Type-1: u, v and the set of vertices whose colors in $\Sigma(G^*\pm<u,v>)$ and $\Sigma(G^*)$ are different, i.e., the vertices in Δ.
- Type-2: the set of vertices that are (1) out-neighbors of the type-1 vertices; and (2) not type-1 vertices.

For every vertex w in q, w is processed using the CAN step. As a result, for each type-2 vertex w, its in-neighbors need to be collected by the CC operator because we do not know whether w is a type-2 vertex in CC. Since a type-2 vertex is a neighbor of a type-1 vertex, the 2-hop neighbors of some type-1 vertices need to be explored, which results in the need to include the factor dmax in the time complexity. To eliminate the dmax factor, exploring the neighbors of type-2 vertices should be avoided.

According to the above analysis, we design two early pruning strategies: early color computation and notification pruning. The former focuses on eliminating the dmax factor by improving the CC and AC operators, and the latter focuses on reducing the n_Δ factor by improving the NC operator.

5.1 Early Color Computation

In this subsection, we discuss how to eliminate the dmax factor by improving CC and AC through a compact index, DINC-Index.

Dynamic In-Neighbor Color Index (DINC-Index)**.** A DINC-Index $\mathcal{I}$ contains the following two components:

- Color Counts $\mathcal{I}.\mathsf{cnt}_u(c)$: the number of u's in-neighbors whose color is c for each vertex $u \in V(G^*)$ and color $c \leq \mathsf{deg}^-(u)$.
- Recolor Candidates $\mathcal{I}.C_u$: the set of colors that are smaller than u.color and not assigned to any in-neighbor of u.

The rationale behind the DINC-Index is as follows. First, for any vertex u in an OCG G^*, we have $u.\mathsf{color} \leq \mathsf{deg}^-(u)$. Therefore, we can uniquely determine the color of u if we know all the colors of u's in-neighbors that are not greater than $\mathsf{deg}^-(u)$. As a result, it is adequate to maintain the color counts for $c \leq \mathsf{deg}^-(u)$ in $\mathcal{I}.\mathsf{cnt}_u(c)$ for our goal. This property is the key to bound the space consumption of the DINC-Index. Based on the definition of the DINC-Index, we can easily derive the following equation:

$$\mathcal{I}.C_u = \{c|c < u.\mathsf{color}, \mathcal{I}.\mathsf{cnt}_u(c) = 0\} \tag{2}$$

According to Eq. 2, for an OCG G^*, a graph coloring is $\Sigma(G^*)$ if and only if $\mathcal{I}.C_u = \emptyset$ for all $u \in V(G^*)$. Therefore, given an OCG G^*, after an edge insertion/deletion, a vertex u changes its color in a certain CAN step if and only if either $\mathcal{I}.C_u \neq \emptyset$ or $\mathcal{I}.\mathsf{cnt}_u(u.\mathsf{color}) \neq 0$. If $\mathcal{I}.C_u \neq \emptyset$, the new color of u can be computed as $u.\mathsf{color} = \min\{c|c \in \mathcal{I}.C_u\}$. If $\mathcal{I}.\mathsf{cnt}_u(u.\mathsf{color}) \neq 0$, the new color of u can be computed as $u.\mathsf{color} = \min\{c|c \in \mathbb{N}, \mathcal{I}.\mathsf{cnt}_u(c) = 0\}$. Consequently, if we can maintain the DINC-Index, we can determine whether a vertex u will change its color in a CAN step in $O(1)$ time, which means we can avoid exploring the neighbors of type-2 vertices. If u does change its color, the new color of u can be computed using $\mathcal{I}.C_u$ and $\mathcal{I}.\mathsf{cnt}_u$. Next, we show how to maintain the DINC-Index without affecting the overall time complexity.

The DINC-Index **Maintenance.** The algorithm to maintain the DINC-Index $\mathcal{I}$ is shown in Algorithm 5. We first introduce two procedures Color-Ins and Color-Dec to maintain $\mathcal{I}.\mathsf{cnt}_u(c)$ and $\mathcal{I}.C_u$ by inserting and deleting a color c in the DINC-Index for vertex u. Color-Ins is shown in line 1-4. Based on the definition of DINC-Index, we only consider the case of $c \leq \mathsf{deg}^-(u)$ (line 2). In this case, we increase $\mathcal{I}.\mathsf{cnt}_u(c)$ by 1 (line 3). As we can guarantee that $\mathcal{I}.\mathsf{cnt}_u(c) \neq 0$, we remove c from $\mathcal{I}.C_u$ if $c \in \mathcal{I}.C_u$ according to Eq. 2. Similarly, in Color-Dec (line 5-8), if $c \leq \mathsf{deg}^-(u)$ (line 6), we first decrease $\mathcal{I}.\mathsf{cnt}_u(c)$ by 1 (line 7), and if $\mathcal{I}.\mathsf{cnt}_u(c) = 0$ and $c < u.\mathsf{color}$, we add c into $\mathcal{I}.C_u$ according to Eq. 2 (line 8). Clearly, the time complexity for both Color-Ins and Color-Dec is $O(1)$. Below we introduce the procedures to maintain the DINC-Index $\mathcal{I}$.

Algorithm 5 DINC-Index Maintenance

```
1: Procedure Color-Ins(DINC-Index $\mathcal{I}$, Vertex $u$, Color $c$)
2: if $c \leq \mathsf{deg}^-(u)$ then
3:     $\mathcal{I}.\mathsf{cnt}_u(c) \leftarrow \mathcal{I}.\mathsf{cnt}_u(c) + 1$;
4:     if $c \in \mathcal{I}.C_u$ then $\mathcal{I}.C_u \leftarrow \mathcal{I}.C_u \setminus \{c\}$;
5: Procedure Color-Dec(DINC-Index $\mathcal{I}$, Vertex $u$, Color $c$)
6: if $c \leq \mathsf{deg}^-(u)$ then
7:     $\mathcal{I}.\mathsf{cnt}_u(c) \leftarrow \mathcal{I}.\mathsf{cnt}_u(c) - 1$;
8:     if $\mathcal{I}.\mathsf{cnt}_u(c) = 0$ and $c < u.\mathsf{color}$ then $\mathcal{I}.C_u \leftarrow \mathcal{I}.C_u \cup \{c\}$;
9: Procedure DINC-Index-Ins(OCG $G^*$, DINC-Index $\mathcal{I}$, Edge $<u, v>$)
10: $\mathbf{S} \leftarrow \emptyset$; $\mathbf{S} \leftarrow \mathbf{S} \cup u$; $\mathbf{S} \leftarrow \mathbf{S} \cup v$; add edge $<u, v>$ in $G^*$;
11: for each $u' \in \mathsf{nbr}^-(u)$ do
12:     if $u \prec u'$ then
13:         remove edge $<u', u>$ and add edge $<u, u'>$ in $G^*$; $\mathbf{S} \leftarrow \mathbf{S} \cup u'$;
14:         Color-Ins($\mathcal{I}$, $u'$, $u$.color); Color-Dec($\mathcal{I}$, $u$, $u'$.color);
15: process line 11-14 by replacing $u$ with $v$ and $u'$ with $v'$;
16: Color-Ins($\mathcal{I}$, $v$, $u$.color);
17: for each $w \in \mathsf{nbr}^-(v)$ do
18:     if $w.\mathsf{color} = \mathsf{deg}^-(v)$ then
19:         Color-Ins($\mathcal{I}$, $v$, $w$.color);
20: return $\mathbf{S}$;
21: Procedure DINC-Index-Del(OCG $G^*$, DINC-Index $\mathcal{I}$, Edge $<u, v>$)
22: $\mathbf{S} \leftarrow \emptyset$; $\mathbf{S} \leftarrow \mathbf{S} \cup u$; $\mathbf{S} \leftarrow \mathbf{S} \cup v$; remove edge $<u, v>$ in $G^*$;
23: for each $u' \in \mathsf{nbr}^+(u)$ do
24:     if $u' \prec u$ then
25:         remove edge $<u, u'>$ and add edge $<u', u>$ in $G^*$; $\mathbf{S} \leftarrow \mathbf{S} \cup u'$;
26:         Color-Ins($\mathcal{I}$, $u$, $u'$.color); Color-Dec($\mathcal{I}$, $u'$, $u$.color);
27: process line 23-26 by replacing $u$ with $v$ and $u'$ with $v'$;
28: Color-Dec($\mathcal{I}$, $v$, $u$.color);
29: $\mathcal{I}.\mathsf{cnt}_v(\mathsf{deg}^-(v) + 1) \leftarrow 0$;
30: return $\mathbf{S}$;
```

Procedure DINC-Index-Ins maintains the DINC-Index $\mathcal{I}$ and the OCG G^* when an edge $<u, v>$ is inserted, which is shown in line 9-20. Line 10-15 is similar to the OCG-Maintain procedure in Algorithm 3. The only difference is that we insert the color u.color to the DINC-Index for u' and delete the color u'.color from the DINC-Index for u for each edge $<u', u>$ that needs to be reversed in G^* to maintain the DINC-Index (line 14). In line 16, we insert u.color to the DINC-Index for v as the edge $<u, v>$ is inserted. Line 17-19 handles a special case: since we only consider the colors $c \leq \mathsf{deg}^-(v)$ in the DINC-Index for v, after inserting $<u, v>$, $\mathsf{deg}^-(v)$ increases by 1, so we should add all vertices in $\mathsf{nbr}^-(v)$ whose color is $\mathsf{deg}^-(v)$ to the DINC-Index for v. Procedure DINC-Index-Del handles the deletion of an edge $<u, v>$ (line 21-30). Line 22-27 follows a similar method to line 10-15 to maintain the OCG G^* and adjust the DINC-Index by considering the reversed edges. Line 28 deletes u.color from the DINC-Index for v because of the deletion of $<u, v>$. In line 29, since $\mathsf{deg}^-(v)$ decreases by 1 and we only consider the colors $c \leq \mathsf{deg}^-(v)$ in $\mathcal{I}.\mathsf{cnt}_v$, we simply set $\mathcal{I}.\mathsf{cnt}_v(\mathsf{deg}^-(v) + 1)$ to be 0.

5.2 Notification Pruning

In this subsection, we explore pruning rules to improve NC. Specifically, when a vertex u's color changes, we aim to find rules that can guarantee that the color of u's neighbor v is not affected by the color change of u, and thus we do not need to push v into q.

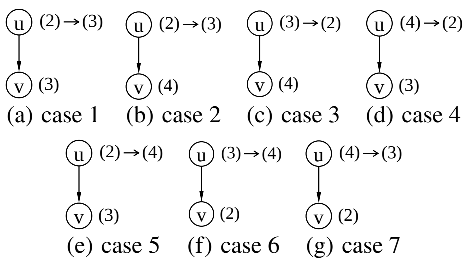


Figure 6: Notification Pruning

In Fig. 6, we consider different cases when the color of a vertex u changes and show how the change of u's color affects the color of its out-neighbor v. In Fig. 6, the colors of u and v before a CAN step are shown in the parentheses near the vertices. In a CAN step, we suppose that the color of u changes. The color change of u is shown near it. For example, in Fig. 6 (a), (2) → (3) means the color of u changes from 2 to 3. For ease of presentation, we use u.old and u.color to represent the colors of u before and after the CAN step and we use v.color to represent the color of v.

We consider different cases based on the relationship among u.old, u.color and v.color. If u.color = v.color, then the color of v has to be reassigned as its color conflicts with u's. Therefore,

◇ **case 1:** u.color = v.color, which is shown in Fig. 6 (a). In this case, v has to be recolored.

Now we consider the cases in which u.color ≠ v.color. We first consider the cases in which u.color < v.color and we have:

◇ **case 2:** u.old < u.color, which is shown in Fig. 6 (b). When the color of u changes from 2 to 3, it is possible for v to be recolored to color 2.

◇ **case 3:** u.color < u.old < v.color, which is shown in Fig. 6 (c). When the color of u changes from 3 to 2, it is possible for v to be recolored with color 3.

◇ **case 4:** u.old > v.color, which is shown in Fig. 6 (d). In this case, the color of u is changed from 4 to 2. The color of v is 3, which means that colors 0, 1, and 2 have been assigned to v's other in-neighbors. Therefore, we cannot find a possible smaller color for v. As a result, the color change of u does not lead to the color change of v in this case.

The cases in which u.color > v.color are shown in Fig. 6 (e)-(g). Summarizing the above cases, we find that when u.color ≠ v.color, whether the color change of u affects the color change of v only depends on the relation between u.old and v.color. If u.old < v.color (cases 2, 3, 5), it is possible that the color of v changes; otherwise (cases 4, 6, 7), the color of v is not affected by the color change of u. Therefore, we have the following three rules to determine whether v should be notified by adding it to q.

Rule 1 If u.color = v.color, v should be notified for recoloring;

Rule 2 If u.color ≠ v.color and u.old < v.color, v should be notified for recoloring;

Rule 3 If u.color ≠ v.color and u.old > v.color, v does not need to be notified for recoloring.

5.3 Early Pruning Dynamic Graph Coloring

Our DC* algorithm, which is integrated with the DINC-Index and notification pruning rules, is shown in Algorithm 6. It follows a similar framework to Algorithm 4. DC*-Ins/DC*-Del handles the edge insertion/deletion, respectively. In DC*-Ins, when an edge is inserted, we first initialize q to be ∅ and maintain G^* and DINC-Index (line 2-3). Line 4 pushes all seed vertices into q and line 5 recolors the vertices using color propagation by invoking a new algorithm

Algorithm 6 DC*(OCG G^*)

```
1: Procedure DC*-Ins(OCG $G^*$, Edge<$u$, $v$>)
2: PriorityQueue $q \leftarrow \emptyset$;
3: S ← DINC-Index-Ins($G^*$, $\mathcal{I}$, <$u$, $v$>);
4: for each $w \in$ S do $q$.push($w$);
5: CAN*($G^*$, $q$, $\mathcal{I}$);
6: Procedure DC*-Del(OCG $G^*$, Edge<$u$, $v$>)
7: PriorityQueue $q \leftarrow \emptyset$;
8: S ← DINC-Index-Del($G^*$, $\mathcal{I}$, <$u$, $v$>);
9: for each $w \in$ S do $q$.push($w$);
10: CAN*($G^*$, $q$, $\mathcal{I}$);
11: Procedure CAN*(OCG $G^*$, PriorityQueue $q$, DINC-Index $\mathcal{I}$)
12: while $q \neq \emptyset$ do
13:     $u \leftarrow q$.pop();
14:     $c_{\text{new}}$ ← CollectColor*($u$);
15:     if $c_{\text{new}} \neq \emptyset$ then
16:         $c_{\text{old}} \leftarrow u$.color;
17:         AssignColor*($\mathcal{I}$, $u$, $c_{\text{new}}$);
18:         NotifyColor*($u$, $c_{\text{old}}$, $q$);
19: Procedure CollectColor*(DINC-Index $\mathcal{I}$, Vertex $u$)
20: if $\mathcal{I}.C_u \neq \emptyset$ then return $\min\{c | c \in \mathcal{I}.C_u\}$;
21: if $\mathcal{I}.\text{cnt}_u(u.\text{color}) \neq 0$ then return $\min\{c | c \in \mathbb{N}, \mathcal{I}.\text{cnt}_u(c) = 0\}$;
22: return ∅;
23: Procedure AssignColor*(DINC-Index $\mathcal{I}$, Vertex $u$, Color $c_{\text{new}}$)
24: for each $v \in \text{nbr}^+(u)$ do
25:     Color-Dec($\mathcal{I}$, $v$, $u$.color); Color-Ins($\mathcal{I}$, $v$, $c_{\text{new}}$);
26: $u$.color ← $c_{\text{new}}$; $\mathcal{I}.C_u \leftarrow \emptyset$;
27: Procedure NotifyColor*(Vertex $u$, Color $c_{\text{old}}$, PriorityQueue $q$)
28: for each $v \in \text{nbr}^+(u)$ do
29:     if $v \notin q$ and ($u$.color = $v$.color or $c_{\text{old}} < v$.color) then
30:         $q$.push($v$);
```

CAN* which is the optimized CAN algorithm using early pruning. The DC*-Del procedure follows a similar framework as DC*-Ins.

The CAN* algorithm is shown in line 11-18. It follows a similar framework as CAN but uses the improved CC, AC, and NC, which are implemented as CollectColor*, AssignColor*, and NotifyColor* respectively. For each vertex u in q, CollectColor* returns the new color of u if its color changes, and ∅ otherwise (line 14). And AssignColor* and NotifyColor* are invoked only if the color of u changes (line 15-18). AssignColor* assigns u with the new color. For NotifyColor*, it takes the old and the new color of u to determine whether an out-neighbor of u needs to be pushed into q.

The new CC, AC and NC are implemented as CollectColor*, AssignColor* and NotifyColor*, respectively. In CollectColor*, if $\mathcal{I}.C_u \neq \emptyset$, we return $\min\{c|c \in \mathcal{I}.C_u\}$ (line 20); otherwise, if $\mathcal{I}.\text{cnt}_u(u.\text{color}) \neq 0$, we return $\min\{c|c \in \mathbb{N}, \mathcal{I}.\text{cnt}_u(c) = 0\}$ (line 21); otherwise, we return ∅ which indicates that the color of u is not changed after the CAN step (line 22). In AssignColor*, we first remove u's old color, and insert u's new color in the DINC-Index for each out-neighbor of u (line 24-25). We then assign the new color to u and set $\mathcal{I}.C_u$ to be ∅ since u does not need to be recolored again (line 26). In NotifyColor*, for all the out-neighbors v of u (line 28), if v is not in q and v satisfies either rule 1 or rule 2 above, we should notify v by adding v to the priority queue q (line 29-30).

Theorem 5.1: *The space consumption of* DINC-Index *is* $O(m)$. □

Proof: For each vertex u, both $\mathcal{I}.\text{cnt}_u$ and $\mathcal{I}.C_u$ can be bounded by $\text{deg}^-(u)$. Therefore, the total size of the DINC-Index is $O(m)$. □

Theorem 5.2: *The time complexity of Algorithm 6 to handle the insertion/deletion of an edge* <u, v> *is* $O(n_\Delta \cdot \log(n_\Delta))$. □

Proof: Let's consider the edge insertion first. Let η be the number of vertices pushed into q. For an edge insertion, DINC-Index-Ins maintains G^* and DINC-Index in $O(\text{dmax})$ (line 3). The push/pop operation for a priority queue can be finished in $O(1)/O(\log \eta)$ if we

Table 1: Datasets used in Experiments

ID	Dataset G	Type	$\|V(G)\|$	$\|E(G)\|$
D0	*MoiveLens*	Rating	150,433	10,000,054
D1	*AS*	Computer	1,696,415	11,095,298
D2	*Epinion*	Rating	996,744	13,668,320
D3	*Libimseti*	Social	220,970	17,359,346
D4	*Baidu*	Hyperlink	2,141,300	17,794,839
D5	*LastFM*	Interaction	1,085,612	19,150,868
D6	*WikiTalk*	Communication	2,987,535	24,981,163
D7	*Flickr*	Social	2,302,925	33,140,017
D8	*Trec*	Text	2,285,379	151,632,178
D9	*WikiEnglish*	Hyperlink	18,268,992	172,183,984
D10	*PL*	Power-law	1,048,576	15,728,640
D11	*UniDeg*	Uniform-degree	1,048,576	10,485,760

use Fibonacci heap. We push (line 4 and 30)/pop (line 13) η vertices in/from q, which can be finished in $O(\eta \cdot \log \eta)$. For the vertices in q, since we can determine whether a vertex u will change its color in a CAN step in $O(1)$ time with DINC-Index (line 20 and 21), we can avoid exploring the in-neighbors of type-2 vertices in Algorithm 6. Therefore, the total time for CollectColor* and AssignColor* can be bounded by the sum of the number of type-1 vertices and that of type-2 vertices, i.e., $O(\eta)$, in Algorithm 6. Since η can be bounded by n_Δ, the time complexity of DC*-Ins is $O(n_\Delta \log(n_\Delta))$. The edge deletion can be proved similarly. Thus, the theorem holds. □

5.4 Vertex Insertion/Deletion

In this paper, we mainly focus on edge insertion/deletion. However, we can extend our techniques to handle vertex insertion/deletion. Specifically, when a vertex u is inserted, we first assign u with color 0. Then, we insert the edges incident to u into OCG and maintain the DINC-Index by DINC-Index-Ins in Algorithm 5. However, instead of returning **S** for each inserted edge, we return **S** until all the incident edges of u have been inserted. At last, we push the vertices in **S** into priority queue and perform the coloring propagation procedure as edge insertion. When the procedure terminates, we obtain the graph coloring. Vertex deletion can be handled similarly as the vertex insertion by DINC-Index-Del.

6. PERFORMANCE STUDIES

This section presents our experimental results. All experiments are conducted on a machine with an Intel Xeon 2.9 GHz CPU (8 cores) and 32 GB main memory, running Linux.

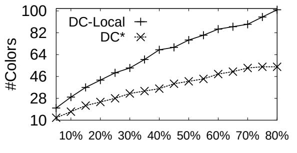
Datasets. We evaluate the algorithms on ten real-world graphs and two synthetic graphs. All the real-world graphs are downloaded from KONECT[1]. For the synthetic graphs, we generate a power-law graph in which edges are randomly added such that the degree distribution follows a power-law distribution and a uniform-degree graph in which all vertices have the same degree [2]. The details of the datasets are shown in Table 1.

Algorithms. We implement and compare four algorithms:

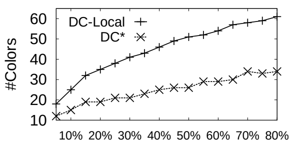
- DC-Local: Algorithm 1 (Section 3).
- DC-Orient: Algorithm 2 (Section 4.2).
- DC-Pri: Algorithm 4 (Section 4.3).
- DC*: Algorithm 6 (Section 5.3).

All algorithms are implemented in C++. The time cost of the algorithms is measured as the amount of elapsed wall-clock time during the program's execution. As all our algorithms use the same number of colors, we only show the number of colors used by DC*.

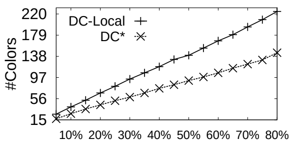
[1]http://konect.uni-koblenz.de/networks
[2]https://networkx.github.io



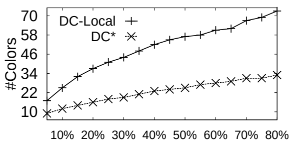
(a) *MoiveLens*



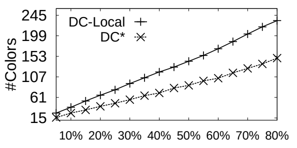
(b) *Libimseti*



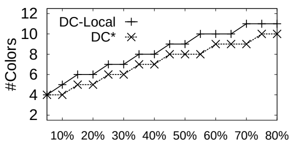
(c) *Flickr*



(d) *Trec*

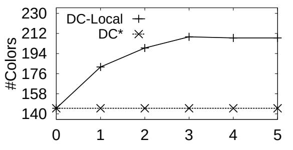


(e) *PL*

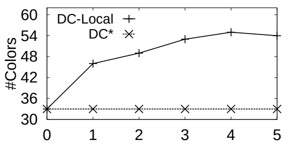


(f) *UniDeg*

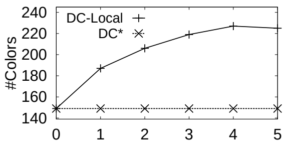
Figure 7: Coloring Quality



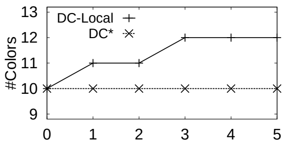
(a) *Flickr*



(b) *Trec*



(c) *PL*



(d) *UniDeg*

Figure 8: Coloring Consistency

When comparing the effectiveness, we only show the representative results as the trends are similar on all datasets.

Exp-1: Coloring Quality. These experiments compare the coloring quality of the four algorithms. To test the coloring quality, we remove all the edges and only keep the vertices for each dataset as the initial graph. Then, we increasingly insert 5% of the edges of the dataset into the initial graph and record the number of colors for each algorithm. The results are shown in Fig. 7.

Fig. 7 shows that: 1) As the percentage of inserted edges increases, the number of colors used by each algorithm also increases. This is because, as the number of edges increases, the relationships among vertices become more complex. As a result, more colors are needed to avoid a color conflict between adjacent vertices. 2) Our algorithm, DC*, uses far fewer colors than DC-Local. This is because DC-Local recolors the graph just based on the local neighbor information while DC* considers the dynamic coloring on a global scope. 3) The difference in the number of used colors between DC-Local and DC* grows larger and larger as the percentage of inserted edges increases. This is also because DC-Local uses local neighbor information while DC* exploits the global information of the graph. As the graph becomes large, the local information increasingly deviates from an optimal solution. Thus, the gap between the number of colors used by them becomes larger and larger as the number of inserted edges increases.

Exp-2: Coloring Consistency. These experiments compare the coloring consistency of the algorithms. We extract 20% the edges

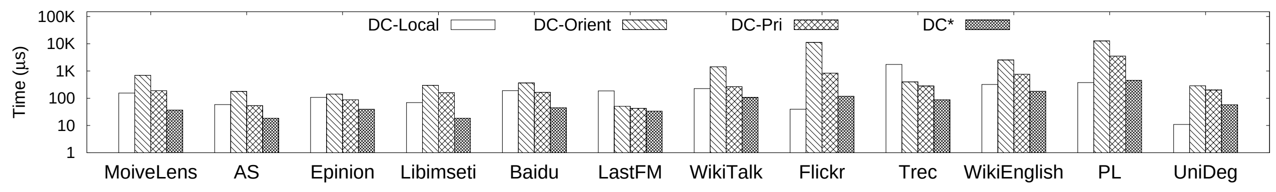


Figure 9: Average Processing Time

from each graph as the pool and use the remaining part as the initial graph. We color the initial graph with Global. To test the coloring consistency, we sample 25% edges from the pool and then randomly insert the sampled edges into the initial graph and delete them. The final graph is the same as the initial graph when the test completes. We conduct the test five times. DC* can obtain the coloring consistency while DC-Local cannot obtain it on all datasets. To further compare them, we record the number of colors used by each algorithm for each test. The result are shown in Fig. 8.

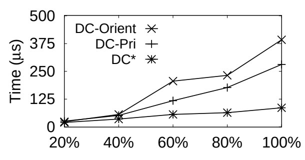
Fig. 8 shows the number of colors used by DC* remains the same on every dataset when the graph is updated. This is because the final graph is the same as the initial graph and DC* can guarantee that it generates the same coloring for a graph regardless of the order in which the edges are inserted/deleted. Conversely, the number of colors used by DC-Local increases sharply at first, then remains stable. The reason for the sharp increase is that DC-Local performs recoloring solely on local information. This leads to a bad coloring compared to our approach when the graph is continuously updated. DC-Local stabilizes after several updating procedures because it can easily find a coloring with a large number of colors.

Exp-3: Processing Time for Each Update. We evaluate the efficiency of the four algorithms in this experiment. We randomly insert and delete 1,000, 000 edges in/from the graph to minimize the effect of caching noises, record the total running time of these updates and compute the average processing time for each update. All the experiments are repeated 5 times. We color the initial graph with the static graph coloring algorithm Global and the preprocessing times for each dataset are 0.8s, 3.2s, 2.2s, 1.9s, 4.8s, 3.2s, 4.1s, 6.1s, 8.7s, 47.6s, 2.5s, 1.9s, respectively. The average processing time for each update is shown in Fig. 9.

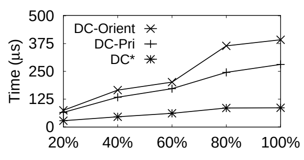
As shown in Fig. 9, among our proposed algorithms, DC-Pri performs better than DC-Orient and DC* has the least average processing time on all datasets. This is because DC-Pri avoids the out-of-order NC problems in DC-Orient and DC* combines the two early pruning strategies. Compared to computing from scratch by Global for each update (the preprocessing time), DC* performs at least three orders of magnitude faster. Considering many graphs are large and frequently updated in real applications, DC* is more practical for the dynamic graph coloring problem. Compared to DC-Local, DC* is more efficient on 9 of the 10 real graphs. For synthetic graphs, DC* consumes more but comparable time than DC-Local on *UniDeg*.

Exp-4: Scalability Testing. We vary $|V|$ and $|E|$ from 20% to 100% of two large datasets *Trec* and *WikiEnglish* to test the scalability of our proposed algorithms. We conduct the experiment the same as Exp-3 on each dataset and the results are shown in Fig. 10.

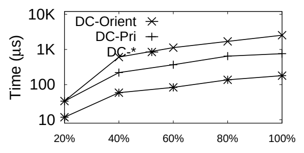
As shown in Fig. 10 (a) and (c), the average processing time of our proposed algorithms for each update increases when $|V|$ increases. This is because as $|V|$ increases, the neighbors for each vertex in the graph generally increases as well. As a result, more vertices need to be explored when an edge is inserted or deleted. Thus, the average processing time increases as $|V|$ increases. Of our proposed algorithms, DC* performs the best on all datasets. This is the result of the combination of the proposed optimization strategies. On all



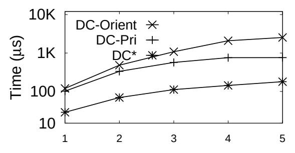
(a) *Trec* (Vary $|V|$)



(b) *Trec* (Vary $|E|$)



(c) *WikiEnglish* (Vary $|V|$)



(d) *WikiEnglish* (Vary $|E|$)

Figure 10: Scalability

Table 2: Static Algorithms on *MoiveLens*

Algorithm	Bktr	HC	TabuCol	AntCol	HEA	ParCol	Global
Time (s)	-	3,013	1,077	-	2,620	5,906	0.8
Colors	-	57	57	-	56	57	65

the datasets, the average processing time of DC* increases stably when $|V|$ increases. Thus, DC* has a good scalability. In Fig. 10 (b) and (d), we can find similar trends when varying $|E|$.

Exp-5: Static Algorithms Testing. In this experiment, we compare the performance of Global with the state-of-the-art static graph coloring algorithms. [43] investigates six representative static graph coloring algorithms in the literature: Bktr, HC, TabuCol, AntCol, HEA and ParCol. We follow [43] and run these algorithms and Global on the graphs used in our experiments. We adopt the same settings as [43] and consider the algorithm cannot finish the test if the algorithm does not terminate in 7,200 seconds or fails due to out of memory exception. The results are shown in Table 2.

For the six algorithms, they cannot finish the test on all datasets except *MoiveLens*. On *MoiveLens*, only HC, TabuCol, HEA and ParCol can finish the test and they are at least three orders of magnitude slower than Global. This is because these algorithms have to search huge candidate space to optimize the coloring while Global only sorts the vertices based on their degrees and traverses the graph once. On the other hand, the number of colors used by these six algorithms and Global are comparable. As real graphs are typically large [25], considering the running time and the number of used colors, we qualify Global as one of the best static graph coloring algorithms for large graphs.

Exp-6: φ for Each Update. Table 3 shows the φ (average number of vertices with color changes) when an edge is inserted/deleted on real graphs. To compute φ, we insert and delete 10,000 edges randomly. When an edge is inserted/deleted, we compute the new coloring with Global and record the number of vertices with changes.

As Table 3 shows, when an edge is inserted/deleted, φ is very small compared to $|V|$ for each dataset. From example, on D1, φ for each update is 9.88 while $|V|$ of D1 is $1,696,415$. Moreover, the maximum φ for the 10 datasets is 40.43 and the average φ for these 10 datasets is only 11.4. Therefore, the number of vertices whose colors are changed in each update is very small in practice.

Table 3: φ for each update

Dataset ID	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9
φ	4.77	9.88	14.86	1.09	10.58	2.01	3.21	24.09	3.27	40.43

Table 4: #Colors of FF, OB and DC*

Dataset ID	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9
FF	106	85	41	64	58	36	116	257	74	78
OB	103	84	39	67	57	35	116	259	74	77
DC*	65	71	21	39	33	25	96	188	37	49

Exp-7: Online Graph Coloring Algorithms Testing. Table 4 shows the number of colors used by two popular online graph coloring algorithms: FF [29] and OB [56] and our algorithm w.r.t vertex insertion on real graphs. The vertices are given in a random order. We perform the testings five times and report the best result.

Table 4 shows that DC* uses much less colors than FF and OB on all datasets. This is because FF and OB determine the color of a new vertex based on the given vertices and the colors of vertices are not allowed to change once assigned while DC* considers the global optimization and the colors of vertices are allowed to update.

7. RELATED WORK

We review the related work from three categories: static graph coloring, online graph coloring and dynamic graph coloring.

Static Graph Coloring. Static graph coloring problem has been explored extensively in the literature. Finding an optimal coloring for a general graph is NP-complete [27]. Zuckerman further shows that for any $\epsilon > 0$, there is no polynomial-time $n^{1-\epsilon}$ approximation algorithm for the optimal graph coloring problem, unless NP=ZPP [72]. Due to the hardness of the problem, many approximation algorithms are proposed, which can be divided into two classes:

(1) Constructive Approaches. A constructive approach builds a coloring step by step by fixing the color of a vertex on each step. Global [68] iterates over the vertices based on their degrees and assigns each vertex the smallest color not assigned to a neighbor. After that, some other variants are proposed, such as [11, 48, 28, 61]. Other constructive methods adopt vertex cut strategy or backtracking strategy, such as [57], [59], and Bktr [40].

(2) Stochastic Search-based Approaches. Stochastic search-based approaches search a space of candidate coloring solutions and attempt to identify the coloring that optimizes a specific objective function. They can be further divided into three subclasses:

The first subclass considers different permutations of the vertices, uses a constructive algorithm to form a feasible solution and chooses the best solution among the permutations. IG [18] generates the permutation iteratively such that the vertices of the independent sets identified in the previous coloring are adjacent in the new permutation. To produce a new permutation, other intelligence algorithm based methods are proposed, such as ant colony optimization algorithm [17], evolutionary based algorithm [24, 52] and hill-climbing algorithm based algorithm (HC) [42].

The second subclass starts by fixing k colours and assigns each vertex one of these colours. As two incident vertices may be assigned with the same color, it makes alterations to reduce the number of such vertices to zero. If this achieves, k decreases; otherwise, k increases. Then, the process restarts with the new k. TabuCol [34] based on tabu search is one of the earliest graph coloring algorithm in this subclass. Other methods using intelligence algorithms have been proposed, including simulated annealing [37], GRASP algorithm [41], local search algorithm [15], variable neighborhood search algorithm [4], evolutionary algorithms [20, 58, 45, 69], HEA [26] and ant colony optimization algorithm AntCol [22].

The third subclass also starts by fixing k colors. It stores the vertices that cannot be feasibly assigned to a color in a set S and aims to find a solution with $S = \emptyset$. Then, it adapts k as the second subclass. An early algorithm in this subclass uses a simulated annealing-based algorithm that operates on a population of candidate solutions [50]. [35] suggests an algorithm that when the search stagnates in one space, it alters the current solution to become a member of another space. [46] hybridises an exploitative local search method with an evolution-based approach. ParCol [7] uses tabu search together with a neighborhood operator to explore the search space.

Online Graph Coloring. Online graph coloring assumes that the vertices with their incident edges are given one by one and a color is assigned to the current vertex before the next vertex is colored. Once a vertex is colored, changes are not allowed [44]. The goal is to minimize the number of used colors. [44] develops a deterministic online algorithm that achieves a competitive factor of $O(n/\log^* n)$. [67] devises a randomized algorithm that attains a competitiveness of $O(n/\sqrt{\log n})$. This bound is improved to $O(n/\log n)$ in [30]. [31] proves that the competitive ratio of any deterministic online algorithm is $\Omega(n/\log^2 n)$. A popular practical online graph coloring algorithm is the first-fit algorithm (FF) [29, 53, 63, 32, 38], which colors the new vertex with the smallest color that is not assigned to its neighbors. OB [56] suggests to assign vertices with the feasible color containing the most vertices. There are considerable studies on online coloring for special graph classes, such as tree [29], interval graph [39], disk graph [13] and bounded treewidth graph [21]. Recent works study scenarios where an online algorithm can query oracle about future information [64, 12, 10].

The Grundy number $\Gamma(G)$ [16] is the maximum k such that G admits a vertex order in which FF yields a proper k-coloring. Thus, $\Gamma(G)$ measures the worst case of FF. Determining $\Gamma(G)$ is NP-hard and testing whether the $\Gamma(G)$ of a given graph is at least k, for a fixed constant k, can be performed in polynomial time [71]. Trivially, $\Gamma(G) \leq \mathsf{dmax} + 1$ but deciding whether $\Gamma(G) \leq \mathsf{dmax}$ is NP-complete [32]. [8] shows an exact algorithm for the Grundy Number with $O(2.443^n)$. Polynomial time algorithms for determining the $\Gamma(G)$ have been proposed for trees [33] and partial k-trees [66]. Recently, [65] gives two upper bounds on $\Gamma(G)$ in terms of its Randic index and clique number.

Dynamic Graph Coloring. There exist several studies on dynamic graph coloring problem in the literature. [9] studies the dynamic graph coloring problem on trees and product graphs and proves various dynamic chromatic number bounds on these types of graphs. [23] studies a decentralized approach for graph coloring problem on vertex-centric distributed systems. DC-Local is the state-of-the-art dynamic graph coloring algorithm [60]. We introduce it in Section 3 and use it as the baseline solution in our experiment.

8. CONCLUSION

In this paper, we study the dynamic graph coloring problem. We propose a color-propagation based algorithm on the oriented coloring graph to bound the explored vertices within the 2-hop neighbors of the vertices in Δ. We further improve our algorithm by devising a novel dynamic in-neighbor color index and some pruning rules. The experimental results demonstrate the effectiveness and efficiency of our approach.

Acknowledgements. Lu Qin is supported by ARC DP160101513. Xuemin Lin is supported by NSFC61232006, ARC DP140103578, DP150102728 and DP170101628. Lijun Chang is supported by ARC DE150100563 and DP160101513. Wenjie Zhang is supported by ARC DP150103071 and DP150102728.

9. REFERENCES

[1] A. Aboulnaga, J. Xiang, and C. Guo. Scalable maximum clique computation using mapreduce. In *Proceedings of ICDE*, pages 74–85, 2013.

[2] D. Alberts, G. Cattaneo, and G. F. Italiano. An empirical study of dynamic graph algorithms. *Journal of Experimental Algorithmics (JEA)*, 2:5, 1997.

[3] N. Armenatzoglou, H. Pham, V. Ntranos, D. Papadias, and C. Shahabi. Real-time multi-criteria social graph partitioning: A game theoretic approach. In *Proceedings of SIGMOD*, pages 1617–1628, 2015.

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

[5] B. Balasundaram and S. Butenko. Graph domination, coloring and cliques in telecommunications. In *Handbook of Optimization in Telecommunications*, pages 865–890. Springer, 2006.

[6] N. Barnier and P. Brisset. Graph coloring for air traffic flow management. *Annals of operations research*, 130(1):163–178, 2004.

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

[8] É. Bonnet, F. Foucaud, E. J. Kim, and F. Sikora. Complexity of grundy coloring and its variants. In *International Computing and Combinatorics Conference*, pages 109–120, 2015.

[9] P. Borowiecki and E. Sidorowicz. Dynamic coloring of graphs. *Fundamenta Informaticae*, 114(2):105–128, 2012.

[10] J. Boyar, L. M. Favrholdt, C. Kudahl, K. S. Larsen, and J. W. Mikkelsen. Online algorithms with advice: A survey. *ACM Computing Surveys (CSUR)*, 50(2):19, 2017.

[11] D. Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22(4):251–256, 1979.

[12] E. Burjons, J. Hromkovič, X. Muñoz, and W. Unger. Online graph coloring with advice and randomized adversary. In *International Conference on Current Trends in Theory and Practice of Informatics*, pages 229–240, 2016.

[13] I. Caragiannis, A. V. Fishkin, C. Kaklamanis, and E. Papaioannou. A tight bound for online colouring of disk graphs. *Theoretical Computer Science*, 384(2-3):152–160, 2007.

[14] A. Carroll, G. Heiser, et al. An analysis of power consumption in a smartphone. In *USENIX annual technical conference*, pages 21–21, 2010.

[15] M. Chiarandini, T. Stützle, et al. An application of iterated local search to graph coloring problem. In *Proceedings of the Computational Symposium on Graph Coloring and its Generalizations*, pages 112–125, 2002.

[16] C. A. Christen and S. M. Selkow. Some perfect coloring properties of graphs. *Journal of Combinatorial Theory, Series B*, 27(1):49–59, 1979.

[17] D. Costa and A. Hertz. Ants can colour graphs. *Journal of the operational research society*, 48(3):295–305, 1997.

[18] J. C. Culberson and F. Luo. Exploring the k-colorable landscape with iterated greedy. *Cliques, coloring, and satisfiability: second DIMACS implementation challenge*, 26:245–284, 1996.

[19] C. Demetrescu, D. Eppstein, Z. Galil, and G. F. Italiano. Dynamic graph algorithms. In *Algorithms and theory of computation handbook*, pages 9–28, 2010.

[20] R. Dorne and J.-K. Hao. A new genetic local search algorithm for graph coloring. In *International Conference on Parallel Problem Solving from Nature*, pages 745–754, 1998.

[21] R. G. Downey and C. McCartin. Online promise problems with online width metrics. *Journal of Computer and System Sciences*, 73(1):57–72, 2007.

[22] K. A. Dowsland and J. M. Thompson. An improved ant colony optimisation heuristic for graph colouring. *Discrete Applied Mathematics*, 156(3):313–324, 2008.

[23] A. Dutot, F. Guinand, D. Olivier, and Y. Pigné. On the decentralized dynamic graph coloring problem. In *Workshop of COSSOM*, 2007.

[24] W. Erben. A grouping genetic algorithm for graph colouring and exam timetabling. In *International Conference on the Practice and Theory of Automated Timetabling*, pages 132–156, 2000.

[25] W. Fan, X. Wang, and Y. Wu. Querying big graphs within bounded resources. In *Proceedings of SIGMOD*, pages 301–312, 2014.

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

[27] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[28] A. H. Gebremedhin, D. Nguyen, M. M. A. Patwary, and A. Pothen. Colpack: Software for graph coloring and related problems in scientific computing. *ACM Transactions on Mathematical Software*, 40(1):1, 2013.

[29] A. Gyárfás and J. Lehel. On-line and first fit colorings of graphs. *Journal of Graph theory*, 12(2):217–227, 1988.

[30] M. M. Halldórsson. Parallel and on-line graph coloring. *Journal of Algorithms*, 23(2):265–280, 1997.

[31] M. M. Halldórsson and M. Szegedy. Lower bounds for on-line graph coloring. In *Proceedings of SODA*, pages 211–216, 1992.

[32] F. Havet and L. Sampaio. On the grundy number of a graph. In *International Symposium on Parameterized and Exact Computation*, pages 170–179, 2010.

[33] S. M. Hedetniemi, S. T. Hedetniemi, and T. Beyer. A linear algorithm for the grundy (coloring) number of a tree. *Congr. Numer*, 36:351–363, 1982.

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

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

[36] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu. Querying k-truss community in large and dynamic graphs. In *Proceedings of SIGMOD*, pages 1311–1322, 2014.

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

[38] H. A. Kierstead, D. A. Smith, and W. T. Trotter. First-fit coloring on interval graphs has performance ratio at least 5. *European Journal of Combinatorics*, 51:236–254, 2016.

[39] H. A. Kierstead and W. T. Trotter. An extremal problem in recursive combinatorics. *Congressus Numerantium*, 33(143-153):98, 1981.

[40] S. M. Korman. The graph-colouring problem. *Combinatorial optimization*, pages 211–235, 1979.

[41] M. Laguna and R. Martí. A grasp for coloring sparse graphs. *Computational optimization and applications*, 19(2):165–178, 2001.

[42] R. Lewis. A general-purpose hill-climbing method for order independent minimum grouping problems: A case study in graph colouring and bin packing. *Computers & Operations Research*, 36(7):2295–2310, 2009.

[43] R. Lewis, J. Thompson, C. Mumford, and J. Gillard. A wide-ranging computational comparison of high-performance graph colouring algorithms. *Computers & Operations Research*, 39(9):1933–1950, 2012.

[44] L. Lovász, M. Saks, and W. T. Trotter. An on-line graph coloring algorithm with sublinear performance ratio. *Discrete Mathematics*, 75(1-3):319–325, 1989.

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

[46] E. Malaguti, M. Monaci, and P. Toth. A metaheuristic approach for the vertex coloring problem. *INFORMS Journal on Computing*, 20(2):302–316, 2008.

[47] J. Manweiler and R. Roy Choudhury. Avoiding the rush hours: Wifi energy management via traffic isolation. In *Proceedings of MobiSys*, pages 253–266, 2011.

[48] D. W. Matula and L. L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *Journal of the ACM*, 30(3):417–427, 1983.

[49] F. Moradi, T. Olovsson, and P. Tsigas. A local seed selection algorithm for overlapping community detection. In *Proceedings of ASONAM*, pages 1–8, 2014.

[50] C. Morgenstern. Distributed coloration neighborhood search. *Discrete Mathematics and Theoretical Computer Science*, 26:335–358, 1996.

[51] A. Mukherjee and M. Hansen. A dynamic rerouting model for air traffic flow management. *Transportation Research Part B: Methodological*, 43(1):159–171, 2009.

[52] C. L. Mumford. New order-based crossovers for the graph coloring problem. In *Parallel Problem Solving from Nature-PPSN IX*, pages 880–889. 2006.

[53] N. Narayanaswamy and R. S. Babu. A note on first-fit coloring of interval graphs. *Order*, 25(1):49–53, 2008.

[54] N. Ohsaka, T. Maehara, and K.-i. Kawarabayashi. Efficient pagerank tracking in evolving networks. In *Proceedings of KDD*, pages 875–884, 2015.

[55] P. R. J. Östergård. A fast algorithm for the maximum clique problem. *Discrete Appl. Math.*, 120(1-3):197–207, 2002.

[56] L. Ouerfelli and H. Bouziri. Greedy algorithms for dynamic graph coloring. In *Proceedings of CCCA*, pages 1–5, 2011.

[57] Y. Peng, B. Choi, B. He, S. Zhou, R. Xu, and X. Yu. Vcolor: A practical vertex-cut based approach for coloring large graphs. In *Proceedings of ICDE*, pages 97–108, 2016.

[58] D. C. Porumbel, J.-K. Hao, and P. Kuntz. An evolutionary approach with diversity guarantee and well-informed grouping recombination for graph coloring. *Computers & Operations Research*, 37(10):1822–1832, 2010.

[59] S. Prestwich. Using an incomplete version of dynamic backtracking for graph colouring. *Electronic Notes in Discrete Mathematics*, 1:61–73, 1999.

[60] D. Preuveneers and Y. Berbers. Acodygra: an agent algorithm for coloring dynamic graphs. *Symbolic and Numeric Algorithms for Scientific Computing*, 6:381–390, 2004.

[61] R. A. Rossi and N. K. Ahmed. Coloring large complex networks. *Social Network Analysis and Mining*, 4(1):228, 2014.

[62] C. Schindelhauer. Mobility in wireless networks. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 100–116, 2006.

[63] S. Smorodinsky. A note on the online first-fit algorithm for coloring k-inductive graphs. *Information Processing Letters*, 109(1):44–45, 2008.

[64] B. C. Steffen. Advice complexity of online graph problems. Ph.D Thesis, 2014.

[65] Z. Tang, B. Wu, L. Hu, and M. Zaker. More bounds for the grundy number of graphs. *Journal of Combinatorial Optimization*, 33(2):580–589, 2017.

[66] J. A. Telle and A. Proskurowski. Algorithms for vertex partitioning problems on partial k-trees. *SIAM Journal on Discrete Mathematics*, 10(4):529–550, 1997.

[67] S. Vishwanathan. Randomized online graph coloring. In *Proceedings of FOCS*, pages 464–469, 1990.

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

[69] Q. Wu and J.-K. Hao. Coloring large graphs based on independent set extraction. *Computers & Operations Research*, 39(2):283–290, 2012.

[70] L. Yuan, L. Qin, X. Lin, L. Chang, and W. Zhang. Diversified top-k clique search. In *Proceedings of ICDE*, pages 387–398, 2015.

[71] M. Zaker. Results on the grundy chromatic number of graphs. *Discrete mathematics*, 306(23):3166–3173, 2006.

[72] D. Zuckerman. Linear degree extractors and the inapproximability of max clique and chromatic number. In *Proceedings of STOC*, pages 681–690, 2006.