Effective and Efficient Dynamic Graph Coloring

Long Yuan[1], Lu Qin[2], Xuemin Lin[1], Lijun Chang[1], and Wenjie Zhang[1]

[1] The University of New South Wales, Australia
{longyuan,lxue,ljchang,zhangw}@cse.unsw.edu.au
[2] University of Technology, Sydney, Australia
lu.qin@uts.edu.au

Technical Report
UNSW-CSE-TR-201707
April 2017

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. Our proposal has 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. To achieve high efficiency, we investigate efficient incremental algorithms to update the graph coloring by exploring a small number of vertices. The algorithms are designed based on the observation that the number of vertices with color changes after a graph update is usually very small. We design a color-propagation based algorithm which only explores the vertices within the 2-hop neighbors of the 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 the color-changed vertices. Moreover, we derive some effective pruning rules to further reduce the number of propagated vertices. The results from extensive performance studies on real and synthetic graphs from various domains demonstrate the high effectiveness and efficiency of our approach.

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 [18].

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 [1, 35].

(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 [9].

(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 problem in a wireless network 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 [39, 8].

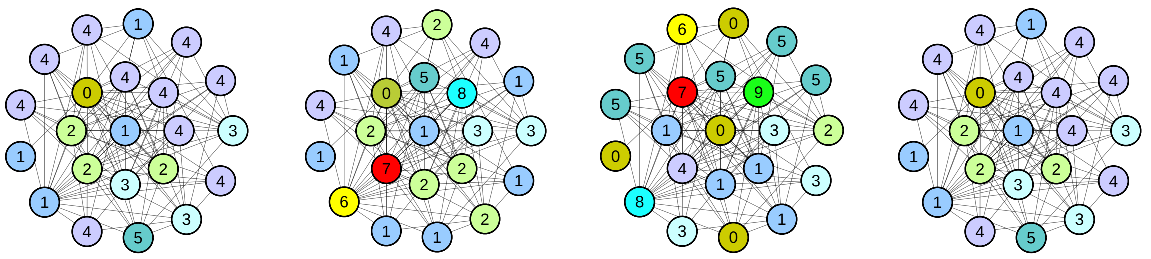
(4) Community Detection in Social Networks. In a social network, graph coloring is used to compute seed vertices that can be expanded to high quality overlapping communities in the network [28].

(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 [31, 45, 2] and graph partitioning [7].

Motivation. In the literature, plenty of algorithms that handle the graph coloring problem in a static graph have been proposed, such as [42, 25, 37, 17, 36, 24, 44, 35]. However, many real-world graphs are highly dynamic [5, 14, 47, 26, 3, 46], 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 [38], 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 in wireless networks [39], power consumption is critical to the usability of mobile devices [12] and WiFi is a prime source of their energy consumption [27]. Consistent coloring result can avoid unnecessary channel changes triggered by the movement of other mobile devices, which would save the power of the mobile devices. In graph partitioning, graph coloring is used as a preprocessing step to classify vertices into different groups [7]. As the groups are classified based on the colors of vertices, consistent coloring result can reduce the repartitioning costs when the graph is updated.

(2) Efficiency. In real applications, many graphs are large and frequently updated. For example, in wireless networks, with the development of transportation facilities,



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

Figure 1.1: Graph coloring on part of the rating network *MoiveLens* (the numbers denote different colors)

the access devices are frequently inserted or removed because of the movement of people [40]; the air traffic flow in air traffic flow networks changes as flights are delayed or cancelled [29]; 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 [30]; Twitter has more than 300 million users and 3 new users join Twitter every second [30]. Therefore, high efficiency is another requirement for a practical dynamic graph coloring algorithm.

In the literature, an algorithm denoted as DC-Local is proposed in [38] for the dynamic graph coloring problem. Briefly, after an edge (u, v) is inserted/deleted in a graph, DC-Local locally updates the graph coloring by adjusting only the colors of vertices u and v and their neighbors in the graph. The time complexity for DC-Local to handle each graph 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, therefore, 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 graph coloring if we obtain the same graph by different edge insertion/deletion orders. The example below illustrates the drawbacks of DC-Local.

Example 1.1: We extract part of a rating network from the *MoiveLens* dataset (`https://movielens.org/`). Initially, we color the network using one of the best static graph coloring algorithms. The coloring result, with 6 colors, is shown in Fig. 1.1 (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. 1.1 (a). The results for the two tests are shown in Fig. 1.1 (b) and Fig. 1.1 (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 clearly shows the 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 two 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. To do this, we investigate one of the best static graph coloring algorithms, Global [42]. 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 (increasing order of their vertex IDs for vertices with the same degree). 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 [31, 45, 2, 7]. To show that our idea is practically applicable, we investigate 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 one of the best static graph coloring algorithms, i.e., 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. 1.1 (a). The initial coloring is computed using Global and the result shown in Fig. 1.1 (d) is exactly the same as the initial coloring in Fig. 1.1 (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 whose colors changed 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-5 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.

Let Δ be the set of vertices with color changes after a graph update. According to the above discussion, $|\Delta|$ is small in practice. Therefore, 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. Therefore, we further propose a dynamic in-neighbor color index $\mathcal{I}$ 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. Such complexity is generally better than the complexity $O(\mathsf{dmax}^2)$ for DC-Local.

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. Our algorithm can achieve high coloring quality and coloring result consistency.

(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 with color changes.

(3) Novel early pruning strategies to further improve the algorithm efficiency. We propose a novel index, called DINC-Index, to efficiently determine whether the color of a vertex will change before color computation occurs and, thus, bound the explored vertices to be within the neighbors of vertices with color changes. 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 from various domains. We conduct extensive performance studies on real and synthetic datasets from various application domains. The experimental results demonstrate that our proposed algorithm can achieve both high effectiveness and high efficiency. Compared to DC-Local, our approach can reduce more than half of the number of colors and is much more efficient than DC-Local in most cases.

Outline. Section 2 provides the problem definition. Section 3 introduces the existing algorithm for dynamic graph coloring. Section 4 analyzes the dynamic graph coloring problem and presents our color propagation based algorithm. Section 5 explores two optimization strategies to further improve the algorithm. Section 6 evaluates our algorithms using extensive experiments. Section 7 reviews the related work and Section 8 concludes the paper.

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 and the number of edges of G by n and m respectively. 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 use $\mathsf{id}(u)$, $\mathsf{nbr}(u)$ and $\mathsf{deg}(u)$ to denote $\mathsf{id}(u, G)$, $\mathsf{nbr}(u, G)$ and $\mathsf{deg}(u, G)$ respectively if the context is self-evident. For 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 $f : V \rightarrow C$ 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 $|\varrho(G)| = \chi(G)$. □

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 insertion and deletion of edges.

Since computing the optimal graph coloring is an NP-hard problem [18], in this paper, we resort to approximate solutions.

Remark. In this paper, we mainly focus on edge insertion/deletion. However, since a vertex insertion/deletion can be regarded as a sequence of edge insertions/deletions preceded/followed by the insertion/deletion of an isolated vertex, our techniques can be directly extended to handle vertex insertions/deletions.

3 The Existing Solution

The state-of-the-art dynamic graph coloring algorithm is proposed in [38]. Before introducing the algorithm, we first define saturation colors as follows.

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., $\mathsf{SC}(u) = \cup_{v \in \mathsf{nbr}(u)}\{v.\mathsf{color}\}$. □

The algorithm 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; otherwise DC-Local does nothing (line 1-7). When an edge (u, v) is deleted, DC-Local recolors both u and v (line 8-10).

To recolor a specific vertex u, DC-Local tries to avoid increasing the number of colors in current coloring based on $\mathsf{SC}(u)$ when recoloring u. Specifically, it first computes the smallest color c_{min} which is not assigned to any neighbor of u (line 12-13). If c_{min} is smaller than the maximum color in $\mathsf{SC}(u)$, c_{min} is assigned to u (line 14-15); otherwise, it first scans each neighbor v of u and computes $\mathsf{SC}(v)$. Then it finds the color c_{cand} which 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 (line 17-20). If c_{cand} is smaller than $c_{\mathsf{min}} - 1$, 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 21-24). Otherwise, it colors u with c_{min} (line 26). For a given vertex u, procedure SmallestUnassignedColor is used to compute the smallest color not assigned to any neighbor of u (line 27-31).

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, $\mathsf{SC}(u)$ can be computed in $O(\mathsf{dmax})$ and procedure DC-Local-Recolor can finish in $O(\mathsf{dmax}^2)$. Thus, for each edge insertion, the time complexity of DC-Local-Ins is $O(\mathsf{dmax}^2)$. The edge deletion can be proved similarly as edge insertion. 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

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

the following two drawbacks:

(D_1) Inferior Coloring Quality. The assumption behind DC-Local is that the graph coloring can be well approximated just by local neighborhood exploration of the vertices in the inserted/deleted edge. However, the assumption does not generally hold in practice since local neighborhood exploration may miss the opportunities to reduce the number of colors globally. As shown in Fig. 1.1, the number of colors used by DC-Local increases from 6 to 10 after a sequence of edge insertions and deletions. The situation is even worse when the graph is becoming large as verified in our experiment. Thus, local neighborhood exploration is inadequate for the dynamic graph coloring problem.

(D_2) Coloring Inconsistency. As shown in Fig. 1.1, the graph coloring generated by DC-Local cannot keep consistent if we obtain the same graph with different edge insertion/deletion orders. This makes the graph coloring generated by DC-Local 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 our new algorithm, the graph is colored from a global perspective rather than using local exploration, and this has the following advantages:

(A_1) High Coloring Quality. Unlike DC-Local, which recolors vertices locally within the neighbors of the vertices in the inserted/deleted edge, our algorithm considers dynamic coloring on a global scale. As a result, our algorithm can achieve the same coloring quality as one of the best static graph coloring algorithms regardless of the number of edge insertion/deletion operations.

(A_2) Coloring Consistency. Additionally, DC-Local may result in inconsistent graph colorings if we get the same graph with different edge insertion/deletion orders. However, the coloring result in our algorithm only depends on the topology of the graph regardless of the edge insertions/deletions order.

(A_3) High Efficiency. Although our algorithm considers recoloring the vertices on a global scale and can achieve the same coloring quality as one of the best static graph coloring algorithms, we do not need to compute the coloring from scratch every time when an edge is inserted/deleted. Instead, our algorithm incrementally updates the graph coloring by only exploring the vertices within the 2-hop neighbors of the vertices with color changes. In practice, very few vertices need color changes; therefore, our algorithm is able to achieve high efficiency.

In this section, we first analyze the dynamic graph coloring problem and propose a basic algorithm that targets A_1 and A_2. Then, we improve the basic algorithm with a prioritized vertex exploration that targets A_3.

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. Since the optimal graph coloring problem is an NP-hard problem [18], and there is no polynomial-time $n^{1-\epsilon}$ approximation algorithm for the optimal graph coloring problem, unless NP=ZPP [48], existing static graph coloring algorithms resort to the greedy approach. One of the best algorithms is Global [42] and it works as follows. It colors vertices in decreasing order of their degrees (the vertices with the same degree are sorted by the increasing order of their ids). For each vertex to be colored, it selects the minimum possible color that is not assigned to its already colored neighbors. The algorithm Global is shown in Algorithm 2.

Algorithm 2 Global(Graph G)

```
1: initialize each vertex $u \in V(G)$ as uncolored;
2: for each $u \in V(G)$ in non-increasing order of deg($u$) do
3:     $u$.color ← GlobalColorV($G, u$);

4: Procedure GlobalColorV($G, u$)
5: $\mathbf{C} \leftarrow \{0, 1, \ldots, \mathsf{deg}(u)\}$, $\mathbb{C} \leftarrow \emptyset$;
6: for each $v \in$ nbr($u$) do
7:     if $v$ is colored then
8:         $\mathbb{C} \leftarrow \mathbb{C} \cup \{v.\mathsf{color}\}$;
9: return $\min\{c | c \in \mathbf{C}, c \notin \mathbb{C}\}$;
```

Algorithm 2 first initializes the vertices in G as uncolored (line 1). Then it iterates over the vertices in non-increasing order of their degrees (increasing order of their vertex ids for vertices with the same degree) and assigns each vertex the color returned by GlobalColorV (line 2-3). For a given vertex u, procedure GlobalColorV is used to compute the smallest color not assigned to a neighbor of u (line 5-9). For a graph G, the time complexity of Algorithm 2 is $O(m + n \log n)$ and it colors G with at most $\mathsf{dmax} + 1$ colors.

Example 4.1: Consider the graph G in Fig. 4.1 (a). To color G, Algorithm 2 first sorts the vertices in G based in their degrees. And the order in which Algorithm 2 colors the vertices is $v_5, v_3, v_4, v_0, v_1, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}$. For v_5, the smallest color not assigned to a neighbor of v_5 is 0, thus GlobalColorV assigns 0 to v_5. Following the order, Algorithm 2 colors all the vertices and the color of each vertex is shown in the parentheses near the vertex in Fig. 4.1 (a). □

Algorithm Global is widely used in the literature due to its high efficiency and good coloring quality in practice [31, 45, 2, 7]. Therefore, we use it to design our approach. The essence of the algorithm Global is to find a coloring f such that the color of every vertex in f 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 $C(u) = \{v.\mathsf{color} \mid v \in \mathsf{nbr}(u) \wedge (\mathsf{deg}(v) > \mathsf{deg}(u) \vee (\mathsf{deg}(v) = \mathsf{deg}(u) \wedge \mathsf{id}(v) < \mathsf{id}(u)))\}$. □

Based on Definition 4.1, the dynamic graph coloring problem can be redefined as follows:

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 insertion and deletion of edges. □

The approach designed based on Definition 4.2 can achieve A_1 and A_2 as the graph coloring satisfying Definition 4.2 is the same as the graph coloring generated by the algorithm Global.

A naive approach to maintain a graph coloring f satisfying Definition 4.2 is to recompute the graph coloring using the algorithm Global for each graph update. Obviously, such an approach is impractical on large graphs. Therefore, we need to design an incremental algorithm to maintain the global color property for all vertices in G. A straightforward solution is to identify the set of vertices that violate the global color property and then recolor these vertices. However, the recoloring for a certain vertex will trigger other vertices to violate the global color property. To efficiently identify the order to recolor vertices, we introduce an auxiliary graph named *oriented coloring graph* in the next subsection.

4.2 Oriented Dynamic Graph Coloring

Oriented Coloring Graph. Oriented coloring graph is constructed based on the total order of vertices which is 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 among 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 $\prec$, which results in a new graph called oriented coloring graph.

Definition 4.4: (Oriented Coloring Graph) Given a graph $G = (V, E)$, the **O**riented **C**oloring **G**raph (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 $<u, v>$ ($<v, u>$). □

Based on the total order $\prec$ used to define the oriented coloring graph, we can easily obtain the following lemma:

Lemma 4.1: *The oriented coloring graph G^* of a graph G is a directed acyclic graph (DAG).* □

Proof: We can prove this by contradiction. Suppose that the OCG G^* of G is not a *DAG*, which means there is a cycle in G^*. Assume that the cycle consists of the directed edges: $<v_1, v_2>,\ldots,<v_{n-1}, v_n>$, $<v_n, v_1>$. According to $<v_n, v_1>$, we can get $v_n \prec v_1$. And according to $<v_1, v_2>,\ldots,<v_{n-1}, v_n>$, we can get $v_1 \prec v_n$. However, based on the definition of $\prec$, it is impossible that for two vertices u and v such that u dominates v and v also dominates u at the same time. Thus, the lemma holds. □

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 use $\mathsf{nbr}^-(u)$, $\mathsf{nbr}^+(u)$, $\mathsf{nbr}(u)$, $\mathsf{deg}^-(u)$, and $\mathsf{deg}^+(u)$ to denote $\mathsf{nbr}^-(u, G^*)$, $\mathsf{nbr}^+(u, G^*)$, $\mathsf{nbr}(u, G^*)$, $\mathsf{deg}^-(u, G^*)$, and $\mathsf{deg}^+(u, G^*)$ respectively 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. We further define the OCG coloring on G^* as follows:

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, we have the following lemma:

Lemma 4.2: *Given a graph G and its* OCG *G^*, f is an* OCG *coloring of G^* if and only if f is a graph coloring of G.* □

Proof: This lemma can be proved by Definition 2.1 and Definition 4.5 directly. □

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 and Lemma 4.2, 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 *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.

Example 4.2: Consider the graph G in Fig. 4.1 (a), the corresponding OCG G^* of G is shown in Fig. 4.1 (b). In G^*, the direction of an edge is decided by the total order $\prec$. For example, as $v_5 \prec v_0$, we create a directed edge $<v_5,v_0>$ in G^*. In Fig. 4.1 (b), we also show $\Sigma(G^*)$. The color of each vertex is shown in the parentheses near the vertex. It is obvious that the color of each vertex in $\Sigma(G^*)$ also satisfies the global color property of

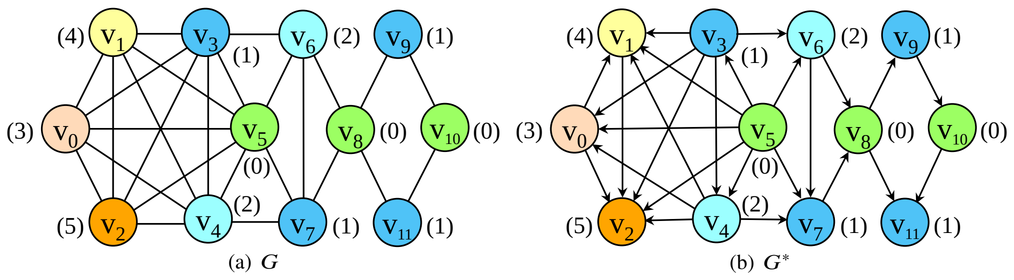


Figure 4.1: Oriented Coloring Graph

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 $\sigma(G^* \pm <u, v>)$. Before showing how to maintain $\Sigma(G^*)$ when G^* is updated, we first introduce the following lemma to reduce the scope of vertices to be recolored:

Lemma 4.3: *Given an* OCG G^* *and* $\Sigma(G^*)$*, when an edge* $<u, v>$ *is inserted/deleted, for a vertex* $w \in V(G^*)$, $w.\mathsf{color}(\Sigma(G^*)) = w.\mathsf{color}(\Sigma(G^* \pm <u, v>))$ *if* $w \prec u$ *in both* G^* *and* $G^* \pm <u, v>$. □

Proof: Based on Definition 4.2, $\Sigma(G^*)$ and $\Sigma(G^* \pm <u, v>)$ is the same as the coloring generated by Global on G^* and $G^* \pm <u, v>$ respectively. Since $w \prec u$ in both G^* and $G^* \pm <u, v>$, which means the coloring order for the vertex w and the vertex colored before w are the same for Global on G^* and $G^* \pm <u, v>$. The lemma holds. □

According to Lemma 4.3, the colors of the vertices which always dominate u before and after the update keep 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:

$$f_{\mathsf{new}}(w) \leftarrow \min\{c | c \in \mathbb{N}, c \notin \cup_{x \in \mathsf{nbr}^-(w)} f_{\mathsf{old}}(x)\} \tag{4.1}$$

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 call f_{new} (f_{old}) as a graph coloring. Based on Eq. 4.1, we have:

Lemma 4.4: *For a given* G^* *and* $\Sigma(G^*)$*, when an edge* $<u, v>$ *is inserted/deleted, the coloring* f *when Eq. 4.1 converges for all vertices* $w \in G^*$ *is* $\Sigma(G^* \pm <u, v>)$. □

Proof: We can 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. 4.1 converges. Thus, the lemma holds. □

According to Lemma 4.4, 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. Lemma 4.3 reduces the scope of vertices to be recolored. However, there are still a large number of vertices to be considered. Below, we introduce a color propagation mechanism on the OCG G^*.

Color Propagation by the CAN Step. According to Eq. 4.1, 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 **C**olors $\mathbb{C}$ 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 **A**ssigns the **C**olor 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 **N**otifies the out-neighbors of u to reassign their **C**olors if b is true. □

A CAN step is defined based on the above three operators:

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. □

According to Lemma 4.1, 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 determine 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 the result, the domination relation between u (v) and their neighbors will change. Consequently, these vertices whose domination relation 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 determine the set of seed vertices for edge insertion and deletion respectively.

Lemma 4.5: *Given an* OCG G^**, after inserting an edge* $<u, v>$*, it is adequate to consider* $\{\{u, v\} \cup I_u \cup I_v\}$ *as 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 $w.\mathsf{color}(\Sigma(G^*)) \neq w.\mathsf{color}(\Sigma(G^* + <u, v>))$, but it is not notified by a CAN step during the color propagation procedure. According to Eq. 4.1, the vertices in $\{\{u, v\} \cup I_u \cup I_v\}$ lead to the color propagation as their in-neighbor are changed in $\Sigma(G^* + <u, v>)$. 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 4.6: *Given an* OCG G^**, after deleting an edge* $<u, v>$*, it is adequate to consider* $\{\{u, v\} \cup D_u \cup D_v\}$ *as 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 4.5. □

With the seed vertices and the color propagation mechanism using the CAN step, we are ready to design our algorithm to maintain $\Sigma(G^*)$ after edge insertion/deletion.

Algorithm Design. Our algorithm DC-Orient to maintain $\Sigma(G^*)$ is shown in Algorithm 3. It contains two main procedures, namely, DC-Orient-Ins and DC-Orient-Del, to handle the edge insertion and deletion respectively. Both DC-Orient-Ins and DC-

Algorithm 3 DC-Orient(OCG G^*)

```
1:  Procedure DC-Orient-Ins(OCG $G^*$, Edge<$u, v$>)
2:    Queue $q \leftarrow \emptyset$;
3:    $\mathbf{S} \leftarrow$ OCG-Ins($G^*$,<$u, v$>) (Algorithm 4);
4:    for each $w \in \mathbf{S}$ 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:    $\mathbf{S} \leftarrow$ OCG-Del($G^*$,<$u, v$>) (Algorithm 4);
10:   for each $w \in \mathbf{S}$ 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:     $\mathbb{C} \leftarrow \mathbb{C} \cup \{v$.color$\}$;
23:   return $\mathbb{C}$;
24: Procedure AssignColor(Vertex $u$, Set $\mathbb{C}$)      //the AC operator
25:   $\mathbf{C} \leftarrow \{0, 1, \ldots, \deg(u)\}$;
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($v$);
```

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 4) to maintain the OCG G^* and obtain the seed vertices in Lemma 4.5/Lemma 4.6 (line 3/line 9). It pushes these vertices into q (line 4-5/line 10-11), and then invokes procedure CAN to iteratively recolor vertices and conduct color propagation by using the CAN step (line 6/line 12).

Procedure CAN 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, it conducts the CC operator (line 16), the AC operator (line 17), and the NC operator (line 18) 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 procedure AssignColor($u, \mathbb{C}$) (line 24-29). According to Definition 4.8, it first computes the smallest color c_{new} which is not in $\mathbb{C}$ (line 25-26). If $c_{\text{new}} \neq u$.color, it assigns c_{new} to u and returns true; otherwise it just

Algorithm 4 OCG-Maintain(OCG G^*)

```
1:  Procedure OCG-Ins(OCG $G^*$, Edge<$u, v$>)
2:  $\mathbf{S} \leftarrow \emptyset$; $\mathbf{S} \leftarrow \mathbf{S} \cup u$; $\mathbf{S} \leftarrow \mathbf{S} \cup v$;
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:          $\mathbf{S} \leftarrow \mathbf{S} \cup u'$;
8:  process line 4-7 by replacing $u$ with $v$ and $u'$ with $v'$;
9:  return $\mathbf{S}$;

10: Procedure OCG-Del(OCG $G^*$, Edge<$u, v$>)
11: $\mathbf{S} \leftarrow \emptyset$; $\mathbf{S} \leftarrow \mathbf{S} \cup u$; $\mathbf{S} \leftarrow \mathbf{S} \cup v$;
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:         $\mathbf{S} \leftarrow \mathbf{S} \cup u'$;
17: process line 13-16 by replacing $u$ with $v$ and $u'$ with $v'$;
18: return $\mathbf{S}$;
```

returns false (line 27-29).

- The NC 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).

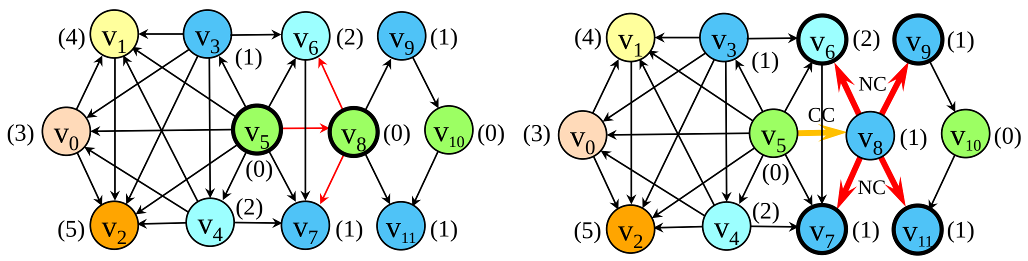
OCG **Maintenance.** OCG-Maintain (Algorithm 4) maintains the OCG G^* and returns the vertices defined in Lemma 4.5/Lemma 4.6. It contains two procedures, namely OCG-Ins and OCG-Del, to handle the edge insertion and deletion respectively.

OCG-Ins uses $\mathbf{S}$ to store the vertices in Lemma 4.5. When an edge <u, v> is inserted, OCG-Ins stores u and v in $\mathbf{S}$ based on Lemma 4.5 (line 2), and inserts edge <u, v> into G^* (line 3). Since 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 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 which 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 domination relation between u and u'. If their domination relation changes after the edge insertion (line 5), OCG-Ins adjusts the direction of the edge (line 6) and adds u' in $\mathbf{S}$ (line 7). Finally, OCG-Ins returns $\mathbf{S}$ in line 9.

Similar to OCG-Ins, $\mathbf{S}$ is used to store the vertices in Lemma 4.6 in OCG-Del. When an edge <u, v> is deleted, OCG-Del stores u and v in $\mathbf{S}$ based on Lemma 4.6 (line 11) and deletes the edge <u, v> from G^* (line 12). After that, it adjusts the direction of edges involving u or v and adds the corresponding vertices to $\mathbf{S}$ according to Lemma 4.6 (line 13-17). Finally, OCG-Del returns $\mathbf{S}$ in line 18.

Example 4.3: Reconsider the OCG G^* in Fig. 4.1 (b) and suppose that an edge <v_5, v_8> is inserted. OCG-Ins first inserts <v_5, v_8> into G^*. After the insertion of <v_5, v_8>, the degree of v_8 increases from 4 to 5. As a result, the domination relationship between v_6 and v_8 is changed. Therefore, OCG-Ins changes <v_6, v_8> to <v_8, v_6> in Fig. 4.2 (a). A similar change is also applied on the directed edge <v_7,v_8>. The changed edges are shown in red lines in Fig. 4.2 (a). OCG-Ins returns the set $\{v_5, v_8, v_6, v_7\}$ based on Lemma 4.5.

Fig. 4.2 (b) shows a CAN step on vertex v_8. It first collects the color set $\mathbb{C}$ of v_8's



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

Figure 4.2: 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 4.3: Steps of DC-Orient for inserting edge $<v_5, v_8>$

in-neighbors by 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. And they will be recolored in the following CAN steps. Such a process terminates when q is empty.

The recoloring procedure of DC-Orient-Ins when $<v_5, v_8>$ is inserted is shown in Fig. 4.3. For each step, we show the vertex on which the CAN step processes, the color of each vertex, and the vertices in q after the CAN step. 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 procedure in 8 steps. The color of each vertex satisfies the oriented global color property after the process. □

Algorithm Analysis. The correctness of Algorithm 3 is shown in the following theorem:

Theorem 4.1: *For a given* OCG *G^* and $\Sigma(G^*)$, when an edge $<u, v>$ is inserted/deleted, the coloring returned by Algorithm 3 is $\Sigma(G^* \pm <u, v>)$.* □

Proof: We consider the edge insertion first. When an edge $<u, v>$ is inserted, in line 3 of Algorithm 3, OCG-Ins can correctly maintain G^* and return the vertices based on Lemma 4.5. In line 16-18 of Algorithm 3, we implement a CAN step. As a CAN step can recolor a vertex based on Eq. 4.1 and in line 14-15 we iteratively process the vertices whose colors may violate $\sigma(G^* + <u, v>)$ until there exists no such kind of vertices. According to Lemma 4.4, when the recoloring procedure converges, the coloring is $\Sigma(G^* + <u, v>)$. Therefore, when an edge $<u, v>$ is inserted, DC-Orient-Ins can return $\Sigma(G^* + <u, v>)$. A similar derivation also holds for the edge deletion. Thus, the theorem holds. □

Since $\Sigma(G^*)$ is only dependent on the topology of graph G^*, it is easy to see that Algorithm 3 can guarantee the coloring consistency.

The time complex of Algorithm 3 is shown below:

Theorem 4.2: *Let n_o be the number of vertices pushed in q in Algorithm 3, the time complexity of Algorithm 3 to handle an edge insertion/deletion is $O(n_o \cdot \mathsf{dmax})$.* □

Proof: Let's consider the edge insertion first. For an edge insertion, DC-Orient-Ins first invokes OCG-Ins to maintain G^* (line 3), which can be finished in $O(\mathsf{dmax})$. In the recoloring procedure (line 6), we push/pop n_o vertices into/from q in line 18/15 and the push/pop operation for a queue can be finished in $O(1)$. Thus, the time complex for this part is $O(n_o)$. 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 this part is $O(n_o \cdot \mathsf{dmax})$. Therefore, the time complex for an edge insertion is $O(n_o \cdot \mathsf{dmax})$. The edge deletion can be proved similarly as edge insertion. Thus, the theorem holds. □

4.3 Prioritized Dynamic Graph Coloring

As shown in Theorem 4.2, the time complexity of DC-Orient-Ins (DC-Orient-Del) depends on the number of vertices pushed in q. However, such a number is not bounded in Algorithm 3 since a vertex may be pushed into q for 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 3 is that the colors of multiple in-neighbors of w are changed, which leads to w to be pushed into q repeatedly by the NC operator. To illustrate this, consider the following scenario: let w_1 and w_2 be two in-neighbors of w. In Algorithm 3, assume that the color of w_1 is changed at a CAN step t_1, then w will be pushed into q as a result of NC(w_1). After that w is popped out from q and recolored at a CAN step t_2. However, the color of w_2 is also changed at a CAN step t_3 after t_2, then w is pushed into q again as a result of NC(w_2).

Example 4.4: The out-of-order NC problem exists in the example in Fig. 4.3. When an edge $<v_5, v_8>$ is inserted in the OCG G^* shown in Fig. 4.1 (b), it uses 8 CAN steps to maintain $\Sigma(G^*)$. Vertex v_{11} has two in-neighbors v_8 and v_{10}. At the CAN step 2, vertex v_8 is recolored and notifies v_{11} to be pushed into q. At the CAN step 6, vertex v_{11} is popped out from q. However, at the 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. For example, in Example 4.4, the vertex v_{11} is recolored at step 6 while its in-neighbors v_{10} is recolored at step 7 that 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 (DAG) according to Lemma 4.1. Therefore, if we follow a topological order of vertices in the DAG 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 direction of edges in G^* are assigned based on 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 5, which contains two procedures, namely, DC-Pri-Ins and DC-Pri-Del, to handle edge insertion

Algorithm 5 DC-Pri(OCG G^*)

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

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

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

and edge deletion respectively. DC-Pri-Ins (DC-Pri-Del) shares a similar framework as 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.

Example 4.5: We still use the OCG G^* in Fig. 4.1 (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 4.3, which is shown in Fig. 4.2 (a). When <v_5, v_8> is inserted, the recoloring procedure of DC-Pri-Ins is shown in Fig. 4.4. 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. Due to 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 while DC-Orient-Ins finishes the procedure in 8 steps. □

Algorithm Analysis. The correctness of Algorithm 5 is shown in the following theorem:

Theorem 4.3: *For a given* OCG G^* *and* $\Sigma(G^*)$*, when an edge* <u, v> *is inserted/deleted, the coloring returned by Algorithm 5 is* $\Sigma(G^* \pm$<u, v>$)$. □

Proof: We prove this theorem by proving that for a vertex w which are pushed into q multiple times in DC-Orient, if we postpone the recoloring of w until all its candidate in-neighbors are recolored, the correctness of DC-Orient still holds. Based on Definition 4.6, the multiple recolorings of w do not affect the colors of its in-neighbors in DC-Orient. Thus, if we postpone the recoloring of w in DC-Orient, for the in-neighbors of w, we can still obtain their colors in $\Sigma(G^* \pm$ <u, v>$)$ correctly. As a result, we can still obtain the color of w in $\Sigma(G^* \pm$ <u, v>$)$ correctly if we postpone its recoloring. We can prove that the colors of w's out-neighbors are not affected by the postponement similarly. Thus, the theorem holds. □

Because of the introduction of vertex priority, we have:

Theorem 4.4: *To handle a certain edge insertion/deletion, each vertex is recolored at* *most once in Algorithm 5.* □

Proof: According to Lemma 4.5, Lemma 4.6 and Definition 4.11, for a vertex u, the colors of its in-neighbors have been decided before recoloring its color. Besides, according to Eq. 4.1, the recoloring of u do not affect its in-neighbors colors. Thus, the colors of u's in-neighbors will not be changed after the color reassignment of u, which means u will not pushed into q again after its recoloring. Thus, the lemma holds. □

We have the following theorem on the number of vertices processed by Algorithm 5:

Theorem 4.5: *For a given* OCG G^**, when an edge* $<u, v>$ *is inserted/deleted, let* Δ *be the set of vertices whose colors in* $\Sigma(G^*)$ *and* $\Sigma(G^* \pm <u, v>)$ *are different, then the number of vertices pushed in q by Algorithm 5 can be bounded by:*

$$n_\Delta = | \cup_{u \in \Delta} \mathsf{nbr}(u) \cup \Delta| \tag{4.2}$$

Obviously n_Δ *is only related to* Δ. □

Proof: Let η be the number of vertices pushed in q. According to Theorem 4.3, the vertices in Δ are pushed into q in Algorithm 5. Besides, based on Definition 4.10, the neighbors of the vertex in Δ are pushed into q by operator NC. Since in Algorithm 5, we can obtain the correct color for each vertex with recoloring it only once, thus, the vertices pushed into q are just the vertices in Δ together with their out-neighbors, i.e., $\eta = | \cup_{u \in \Delta} \mathsf{nbr}^+(u) \cup \Delta|$. It is obvious that $\eta \le n_\Delta$. Thus, the lemma holds. □

The time complexity of Algorithm 5 is shown below:

Theorem 4.6: *The time complexity of Algorithm 5 to handle an edge insertion/deletion 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 $O(\mathsf{dmax})$. In the recoloring procedure (line 5), we push/pop η vertices into/from q and the push/pop operation for a priority queue can be finished in $O(1)/O(\log \eta)$ if we implement the priority queue as the Fibonacci heap. Thus, the time complex for this part is $O(\eta \cdot \log \eta)$. For each vertex u in q, both CollectColor and AssignColor can be finished in $O(\mathsf{dmax})$. Thus, the time for this part is $O(\eta \cdot \mathsf{dmax})$. Since η can be bounded by n_Δ based on Theorem 4.5, the time complexity of DC-Pri-Ins is $O(n_\Delta(\mathsf{dmax} + \log(n_\Delta)))$. The edge deletion can be proved similarly. Thus, the theorem holds. □

Remark. Comparing to Theorem 4.2, n_Δ in Theorem 4.6 can be well bounded by the number of vertices in Δ and their neighbors according to Theorem 4.5. Note that Δ is the set of vertices whose colors in $\Sigma(G^*)$ and $\Sigma(G^* \pm <u, v>)$ are different, not just u or v. Consequently, Algorithm 5 only explores vertices within the 2-hop neighbors of the vertices in Δ. On the other hand, n_o in Theorem 4.2 cannot be bounded. According to our experiments in Section 6, we have $n_o >> n_\Delta$ in practice. Therefore, Algorithm 5 is a significant improvement of Algorithm 3. □

5 Early Pruning

In this section, we aim to further improve the performance of our algorithm using early pruning strategies.

5.1 Solution Overview

In Theorem 4.6, the time complexity of Algorithm 5 depends on two factors: n_Δ and dmax. Although n_Δ can be well bounded according to Theorem 4.5, dmax can be large. In this section, we try to eliminate the factor dmax from the time complexity and further reduce the factor n_Δ. Below, we show why the factor dmax is involved in the time complexity of DC-Pri. Note that in DC-Pri, two types of vertices are pushed in the priority queue q:

- Type-1: 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, we will process w using the CAN step. As a result, for each type-2 vertex w, we need to collect all its in-neighbors 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 1-hop neighbor of a type-1 vertex. This indicates that we need to explore the 2-hop neighbors of some type-1 vertices, which results in the factor dmax in the time complexity. Based on this, to eliminate the dmax factor, we should avoid exploring neighbors of type-2 vertices. Below, we revisit the three operators to find possible early pruning strategies.

Revisit the CC Operator. In CC, we need to visit all the in-neighbors of a vertex u because we do not know whether u is a type-2 vertex. Note that u's in-neighbors' colors uniquely determine the color of u. Therefore, if we can maintain some useful information of u's in-neighbors' colors based on which whether u is a type-2 vertex can be quickly determined, we can avoid exploring the in-neighbors of u if u is a type-2 vertex.

Revisit the AC Operator. In AC, we compute a minimum possible color for a vertex u by visiting all the colors in a set $\mathbb{C}$. If we know a priori whether u is a type-2 vertex, we can avoid the color computation for those type-2 vertices.

Revisit the NC Operator. In NC, for a type-1 vertex u, we need to push all the out-neighbors of v into q. If we can find some conditions to quickly determine whether the color change of u will cause the color change of v, we can prune v by not pushing it into q, and thus reduce the number of the candidates.

The General Ideas. Based on the above analysis, we design two early pruning strategies, namely, early color computation and notification pruning. The former targets on eliminating dmax factor by improving the CC and AC operators, and the latter targets on reducing the n_Δ factor by improving the NC operator.

- *Early Color Computation.* The general idea of early color computation is to determine whether a vertex u is a type-2 vertex in CC rather than in AC by maintaining some summary information about u's in-neighbors' colors. To do this, we introduce a Dynamic In-Neighbor Colors Index (DINC-Index). With the index, we can determine whether u is a type-2 vertex in $O(1)$ time. The DINC-Index has linear space consumption and can be maintained efficiently. With early color computation, we can eliminate the dmax factor in the time complexity (Theorem 4.6) and completely avoid exploring the neighbors of type-2 vertices.
- *Notification Pruning.* The early color computation effectively reduces the cost spent on exploring the neighbors of type-2 vertices. However, it cannot reduce the number of type-2 vertices. To handle this, we further propose notification pruning, which aims to reduce the number of type-2 vertices by optimizing the NC operator. Briefly speaking, based on some effective pruning rules, when a vertex changes its color, we only need to notify a subset of its out-neighbors rather than the whole out-neighbors to be added into q. With notification pruning, we can reduce the n_Δ factor in the time complexity (Theorem 4.6).

Algorithm 6 DC*(OCG G^*)

```
1: Procedure DC*-Ins(OCG $G^*$, Edge<$u, v$>)
2:   PriorityQueue $q \leftarrow \emptyset$;
3:   $\mathbf{S} \leftarrow$ DINC-Index-Ins($G^*$, $\mathcal{I}$, <$u, v$>); (Algorithm 7)
4:   for each $w \in \mathbf{S}$ 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:   $\mathbf{S} \leftarrow$ DINC-Index-Del($G^*$, $\mathcal{I}$, <$u, v$>); (Algorithm 7)
9:   for each $w \in \mathbf{S}$ 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:      $c_{\text{new}} \leftarrow$ CollectColor*($u$); (Algorithm 7)
15:      if $c_{\text{new}} \neq \emptyset$ then
16:          $c_{\text{old}} \leftarrow u$.color;
17:          AssignColor*($\mathcal{I}, u, c_{\text{new}}$); (Algorithm 7)
18:          NotifyColor*($u, c_{\text{old}}, q$); (Algorithm 9)
```

Algorithm Framework. Our algorithm DC* is shown in Algorithm 6, which follows a similar framework of Algorithm 5. It contains two main procedures DC*-Ins and DC*-Del to handle edge insertion and deletion respectively. In DC*-Ins (line 1-5), when an edge <u, v> is inserted into G^*, we first initialize q to be $\emptyset$. In line 3, we maintain the OCG G^* as well as the DINC-Index $\mathcal{I}$. The details of DINC-Index and its maintenance will be introduced in Section 5.2. Line 4 pushes all seed vertices into q and line 5 recolors the vertices using color propagation by invoking a new algorithm CAN* which is the optimized CAN algorithm using early pruning. The DC*-Del procedure (line 6-10) 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. Here, for each vertex u in q, CollectColor* returns the new color of u if it changes, and $\emptyset$ otherwise (line 14). And AssignColor* and NotifyColor* will be invoked only if the color of u changes (line 15-18). For NotifyColor*, it takes both the old color and the new color of u to determine whether an out-neighbor of u needs to be pushed into q (line 16 and line 18).

5.2 Early Color Computation

In this subsection, we discuss how to improve CC and AC using early color computation. As shown in Section 5.1, we design a Dynamic In-Neighbor Colors Index (DINC-Index) to maintain some summary information about the in-neighbors' colors for each vertex.

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

- Color Counts $\mathcal{I}.\text{cnt}_u(c)$: the number of u's in-neighbors whose color is c for each vertex $u \in V(G^*)$ and color $c \leq \text{deg}^-(u)$.
- 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, it is adequate to maintain the color counts for $c \leq \text{deg}^-(u)$ in $\mathcal{I}.\text{cnt}_u(c)$ for our goal, which is based on the following lemma:

Lemma 5.1: *Given an* OCG G^* *and the graph coloring* $\Sigma(G^*)$*, for any vertex u, we have* $u.\mathsf{color} \leq \mathsf{deg}^-(u)$. □

Proof: This lemma can be proved by Definition 4.6 directly. □

According to Lemma 5.1, we can uniquely determine the color of u using $\mathcal{I}.\mathsf{cnt}_u(c)$ for all $c \leq \mathsf{deg}^-(u)$. Therefore, we do not need to maintain the color counts for those colors $c > \mathsf{deg}^-(u)$. This property is the key to bound the space consumption of the DINC-Index. Based on Lemma 5.1 and the definition of the DINC-Index, we can easily derive the following equation:

$$\mathcal{I}.C_u = \{c | c < u.\mathsf{color}, \mathcal{I}.\mathsf{cnt}_u(c) = 0\} \tag{5.1}$$

It is easy to derive the following lemma regarding $\mathcal{I}.C_u$.

Lemma 5.2: *Given an* OCG G^**, a graph coloring is* $\Sigma(G^*)$ *if and only if* $\mathcal{I}.C_u = \emptyset$ *for all* $u \in V(G^*)$. □

Proof: This lemma can be proved by Definition 4.6 and Eq. 5.1 directly. □

Based on Lemma 5.2, we can easily derive the following lemma.

Lemma 5.3: *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* $\mathcal{I}.\mathsf{cnt}_u(u.\mathsf{color}) \neq 0$.

(1) If $\mathcal{I}.C_u \neq \emptyset$*, the new color of u can be computed as*

$$u.\mathsf{color} = \min\{c | c \in \mathcal{I}.C_u\};$$

(2) If $\mathcal{I}.\mathsf{cnt}_u(u.\mathsf{color}) \neq 0$*, the new color of u can be computed as*

$$u.\mathsf{color} = \min\{c | c \in \mathbb{N}, \mathcal{I}.cnt_u(c) = 0\}.$$ □

Proof: Based on the definition of DINC-Index, if $\mathcal{I}.C_u \neq \emptyset$, then the colors in $\mathcal{I}.C_u$ are the colors which are assigned to u's in-neighbor before the update but not assigned to any in-neighbor of u after the update. Based on Definition 4.6, u has to change its color to the minimum color in $\mathcal{I}.C_u$; if $\mathcal{I}.C_u = \emptyset$ but $\mathcal{I}.\mathsf{cnt}_u(u.\mathsf{color}) \neq 0$, which means all the colors that are not bigger than $u.\mathsf{color}$ are assigned to u's in-neighbors after the update. Then, we have to reassign u's color based on Definition 4.6; Otherwise, based on Definition 4.6, the color of u satisfies $\sigma(G^* \pm <u, v>)$ and its color does not need to change. Thus, the lemma holds. □

Based on Lemma 5.3, 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. And if u will change its color, we can compute the new color of u 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 7. 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. 5.1. 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. 5.1 (line 8). Obviously, the time complexity for both Color-Ins and Color-Dec is $O(1)$. Below we introduce the procedures to maintain the DINC-Index $\mathcal{I}$.

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 procedure OCG-Maintain in Algorithm 4. The only difference is that, for each edge

Algorithm 7 DINC-Index Maintenance

```
1: Procedure Color-Ins(DINC-Index $\mathcal{I}$, Vertex $u$, Color $c$)
2: if $c \le \mathsf{deg}^-(u)$ 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 $c \le \mathsf{deg}^-(u)$ 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 $\mathbf{S}$;
```

$<u', u>$ that needs to be reversed in G^*, we insert the color u.color to the DINC-Index for u' and delete the color u'.color from the DINC-Index for u (line 14) to maintain the DINC-Index. 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 $c \le \mathsf{deg}^-(v)$ 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 way as 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 due to the deletion of $<u, v>$. In line 29, since $\mathsf{deg}^-(v)$ decreases by 1 and we only consider the colors $c \le \mathsf{deg}^-(v)$ in $\mathcal{I}.\mathsf{cnt}_v$, we simply set $\mathcal{I}.\mathsf{cnt}_v(\mathsf{deg}^-(v) + 1)$ to be 0.

Algorithm Design. The new operators CC and AC are implemented as CollectColor* and AssignColor* respectively, which are shown in Algorithm 8. In CollectColor* (line 1-4), according to Lemma 5.3, if $\mathcal{I}.C_u \neq \emptyset$, we return $\min\{c|c \in \mathcal{I}.C_u\}$ (line 2); otherwise, if $\mathcal{I}.\mathsf{cnt}_u(u.\mathsf{color}) \neq 0$, we return $\min\{c|c \in \mathbb{N}, \mathcal{I}.\mathsf{cnt}_u(c) = 0\}$ (line 3); otherwise, we return $\emptyset$ which indicates that the color of u is not changed after the CAN step (line 4). In AssignColor* (line 5-8), we first remove the old color of u, and insert the new color of u in the DINC-Index for each out-neighbor of u (line 6-7). We then assign the new color to u and set $\mathcal{I}.C_u$ to be $\emptyset$ since u does not need to be recolored

Algorithm 8 Early Color Computation with DINC-Index

```
1: Procedure CollectColor*(DINC-Index $\mathcal{I}$, Vertex $u$)
2:   if $\mathcal{I}.C_u \neq \emptyset$ then return $\min\{c|c \in \mathcal{I}.C_u\}$;
3:   if $\mathcal{I}.\mathsf{cnt}_u(u.\mathsf{color}) \neq 0$ then return $\min\{c|c \in \mathbb{N}, \mathcal{I}.\mathsf{cnt}_u(c) = 0\}$;
4:   return $\emptyset$;
5: Procedure AssignColor*(DINC-Index $\mathcal{I}$, Vertex $u$, Color $c_{\mathsf{new}}$)
6: for each $v \in \mathsf{nbr}^+(u)$ do
7:     Color-Dec($\mathcal{I}$, $v$, $u.\mathsf{color}$); Color-Ins($\mathcal{I}$, $v$, $c_{\mathsf{new}}$);
8: $u.\mathsf{color} \leftarrow c_{\mathsf{new}}$; $\mathcal{I}.C_u \leftarrow \emptyset$;
```

again (line 8).

Algorithm Analysis. The space complexity of the DINC-Index is shown in the following theorem:

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

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

The time complexity for the algorithm with DINC-Index is:

Theorem 5.2: *The time complexity of the algorithm with* DINC-Index *to handle an edge insertion/deletion is* $O(n_\Delta \cdot \log(n_\Delta))$. □

Proof: This theorem can be proved similarly as Theorem 4.6. The only difference is that we can determine whether a vertex u will change its color in a CAN step in $O(1)$ time with DINC-Index. Therefore, in a recoloring procedure of DC-Pri, we have to explore the in-neighbors of type-2 vertices while with DINC-Index, this part of exploration can be totally avoid in the algorithm. As a result, in the recoloring procedure of DC-Pri, the total time for CollectColor and AssignColor can be bounded by $O(n_\Delta \cdot \mathsf{dmax})$ while with DINC-Index, this part can be bounded by $O(n_\Delta)$. Thus, the total time complexity with DINC-Index can be bounded by $O(n_\Delta \log(n_\Delta))$. □

Comparing Theorem 5.2 with Theorem 4.6, the factor dmax is eliminated from the time complexity. Therefore, the algorithm with DINC-Index for early color computation is more efficient than algorithm DC-Pri (Algorithm 5).

5.3 Notification Pruning

In this subsection, we explore pruning rules to improve the NC operator. Specifically, when the color of a vertex u changes, for one of its out-neighbors v, we aim to find some rules that can guarantee that the color of v is not affected by the color change of u, and thus we do not need to push v into q.

In Fig. 5.1, 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. 5.1, 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. 5.1 (a), (2) $\longrightarrow$ (3) means the color of u changes from 2 to 3. For ease of presentation, we use $u.\mathsf{old}$ and $u.\mathsf{color}$ to represent the colors of u before and after the CAN step and we use $v.\mathsf{color}$ to represent the color of v. We consider different cases to show how the change of u's color affects the color of v.

We consider different cases based on the relationship among $u.\mathsf{old}$, $u.\mathsf{color}$ and $v.\mathsf{color}$. A direct classification criteria is based on the relation between $u.\mathsf{color}$ and $v.\mathsf{color}$. If $u.\mathsf{color} = v.\mathsf{color}$, then the color of v has to be reassigned as its color conflicts with u's. Therefore, we have the following case:

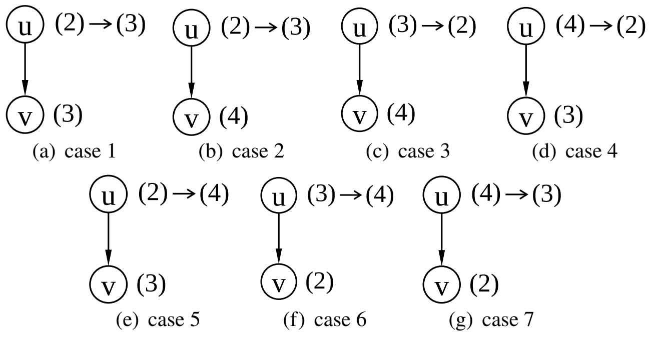


Figure 5.1: Notification Pruning

Algorithm 9 Notification Pruning

```
1: Procedure NotifyColor*(Vertex $u$, Color $c_{\text{old}}$, PriorityQueue $q$)
2: for each $v \in \text{nbr}^+(u)$ do
3:     if $v \notin q$ and ($u$.color = $v$.color or $c_{\text{old}} < v$.color) then
4:         $q$.push($v$);
```

◇ **case 1:** u.color = v.color, which is shown in Fig. 5.1 (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 the following three cases:

◇ **case 2:** u.old < u.color, which is shown in Fig. 5.1 (b). When the color of u changes from 2 to 3, v is possible to be recolored with color 2.

◇ **case 3:** u.color < u.old < v.color, which is shown in Fig. 5.1 (c). When the color of u changes from 3 to 2, v is possible to be recolored with color 3.

◇ **case 4:** u.old > v.color, which is shown in Fig. 5.1 (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 can not 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.

Then we consider the cases in which u.color > v.color, and we have the following three cases:

◇ **case 5:** u.old < v.color, which is shown in Fig. 5.1 (e). When the color of u changes from 2 to 4, v is possible to be recolored with color 2.

◇ **case 6:** v.color < u.old < u.color, which is shown in Fig. 5.1 (f). In this case, the color of u changes from 3 to 4, and the color of v is 2. We cannot find a smaller color for v. Therefore, the color change of u does not lead to the color change of v.

◇ **case 7:** u.old > u.color, which is shown in Fig. 5.1 (g). In this case, the color of u changes from 4 to 3, and the color of v is 2. We cannot find a smaller color for v. Therefore, the color change of u does not lead to the color change of v.

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 $\neq$ v.color and u.old $<$ v.color, v should be notified for recoloring;

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

Algorithm Design and Analysis. Based on the above three rules, the new NC algorithm, which is implemented as NotifyColor*, is shown in Algorithm 9. For all the out-neighbors v of u (line 2), 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 3-4). The time complexity of the final algorithm DC* (Algorithm 6) with all the early pruning strategies is shown in the following theorem.

Theorem 5.3: *The time complexity of Algorithm 6 to handle an edge insertion/deletion is $O(n^*_\Delta \cdot \log(n^*_\Delta))$, where $n^*_\Delta \leq n_\Delta$.* □

Proof: This theorem can be proved similarly as Theorem 5.2. □

Comparing to Theorem 5.2, DC* (Algorithm 6) reduces n_Δ to be n^*_Δ which is usually much smaller than n_Δ in practice.

6 Performance Studies

In this section, we show our experimental results. All of our experiments are conducted on a machine with an Intel Xeon 2.9 GHz CPU (8 cores) and 16 GB main memory, running Linux (Red Hat Enterprise Linux 6.4, 64bit).

Datasets. We evaluate the algorithms on ten real-world graphs and two synthetic graphs. All the real-world graphs are downloaded from KONECT (`http://konect.uni-koblenz.de/networks/`). For the synthetic graphs, we generate two types of graphs by GTGraph (`http://www.cse.psu.edu/~kxm85/software/GTgraph/`), as follows:

- Power-law graphs: A power-law graph is a random graph in which edges are randomly added such that the degree distribution follows a power-law distribution.
- SSCA: A SSCA graph contains a collection of randomly sized cliques and also random inter-clique edges.

ID	**Dataset G**	**Type**	$\lvert V(G)\rvert$	$\lvert E(G)\rvert$	**Avg Degree**
D0	*MoiveLens*	Rating	150,433	10,000,054	132.95
D1	*AS*	Computer	1,696,415	11,095,298	13.08
D2	*Epinion*	Rating	996,744	13,668,320	27.42
D3	*Libimseti*	Social	220,970	17,359,346	157.11
D4	*Baidu*	Hyperlink	2,141,300	17,794,839	16.62
D5	*LastFM*	Interaction	1,085,612	19,150,868	35.28
D6	*WikiTalk*	Communication	2,987,535	24,981,163	16.72
D7	*Flickr*	Social	2,302,925	33,140,017	28.78
D8	*Trec*	Text	2,285,379	151,632,178	132.69
D9	*WikiEnglish*	Hyperlink	18,268,992	172,183,984	18.84
D10	*PL0*	Power-law	1,048,576	15,728,640	30.00
D11	*SSCA0*	SSCA	1,048,576	30,965,547	59.06

Table 6.1: Datasets used in Experiments

Algorithms. We implement and compare five algorithms:

- DC-Local: Algorithm 1 (Section 3).
- DC-Orient: Algorithm 3 (Section 4.2).
- DC-Pri: Algorithm 5 (Section 4.3).
- DC-Index: DC-Pri + DINC-Index (Section 5.2).

- DC*: Algorithm 6 (Section 5.1).

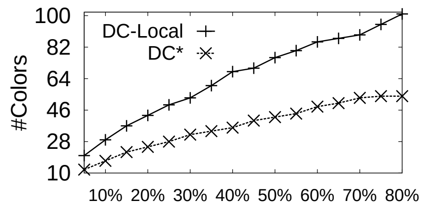
DC-Local is the state-of-the-art dynamic graph coloring algorithm in the literature, which is introduced in Section 3. The remaining algorithms are proposed in this paper. DC-Orient and DC-Pri are presented in Section 4.2 and Section 4.3, respectively. DC-Index combines the early color computation strategy DINC-Index (Section 5.2) to DC-Pri. DC* is our final algorithm and uses all the pruning strategies. All algorithms are implemented in C++. The time cost of algorithms are measured as the amount of wall-clock time elapsed during the program's execution. Since the number of colors used by our algorithms are the same, we only show the number of colors used by DC* when comparing the effectiveness of the algorithms.

Exp-1: Coloring Quality. We compare the coloring quality of the five algorithms in this experiment. To test the coloring quality, we remove all the edges and just keep the vertices for each dateset 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. Fig. 6.1 shows the results on all datasets.

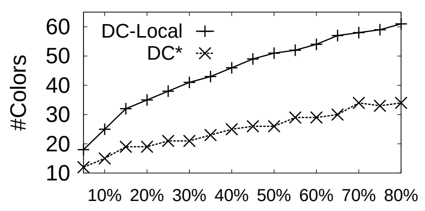
From Fig. 6.1, we can see: 1) as the percentage of inserted edges increases, the number of colors used by each algorithm increases as well. This is because as the number of edges increases, the relations among vertices become more complex. As a result, more colors are needed to avoid color conflict between adjacent vertices. 2) our algorithm DC* uses much less colors than DC-Local. For example, on *Libimseti* (Fig. 6.1(b)), when 80% of the edges are inserted, the number of color used by DC-Local is 64 while that of DC* is only 34. This is because DC-Local recolors the graph just based on the local neighbor information while DC* considers the dynamic coloring in a global scope. 3) the difference on the number of used colors between DC-Local and DC* becomes larger and larger as the percentage of inserted edges increases. For example, on *Trec* (Fig. 6.1 (d)), the difference is 7 when 5% of the edges are inserted while it increases to 37 when 80% of the edges are inserted. This is also because DC-Local uses the local neighbor information while DC* exploits the global information of the graph. As the graph is becoming large, the local information increasingly deviates from optimal solution. Thus, the gap between the number of colors used by these two algorithms becomes larger and larger as the number of inserted edges increases.

Exp-2: Coloring Consistency. In this experiment, we compare the coloring consistency of the algorithms. We extract 20% of the edges from each graph as the edge pool and take the remaining part as the initial graph. We color the initial graph by Global. To test the coloring consistency, for each updating procedure, we sample 25% (5% of the original graph) of edges in the pool and then insert the sampled edges into the initial graph and delete these edges randomly. The final graph is the same as the initial graph when the updating procedure finishes and we record the number of colors used by each algorithm. We conduct the updating procedure five times. Fig. 6.2 shows the results on all datasets.

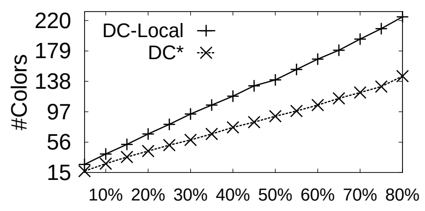
In Fig. 6.2, on every dataset, the number of colors used by DC* keeps the same when the graph is updated. For example, on *Trec* (Fig. 6.2 (d)), the number of used colors is always 33. This is because the final graph is the same as the initial graph for each updating procedure and DC* can guarantee its generated coloring for the same graph is the same regardless of the order in which the edges are inserted/deleted. On the other hand, the number of colors used by DC-Local increases sharply at first and then keeps stable. For example, on *Flickr* (Fig. 6.2 (c)), its number of used color increases from 149 to 209 after 3 updating procedures and flows around 209 afterwards. The reason for the sharp increment is that DC-Local performs the recoloring based on local



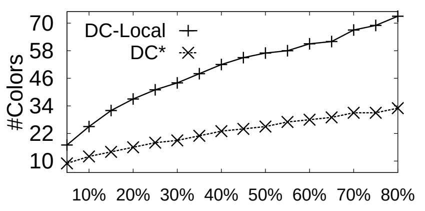
(a) *MoiveLens*



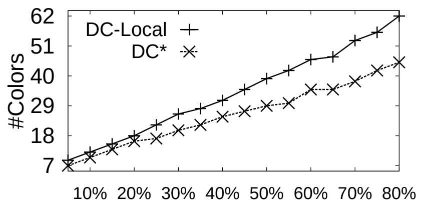
(b) *Libimseti*



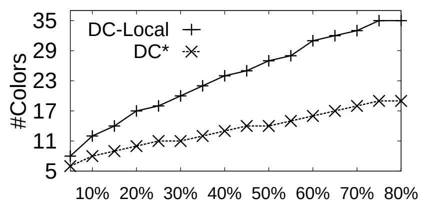
(c) *Flickr*



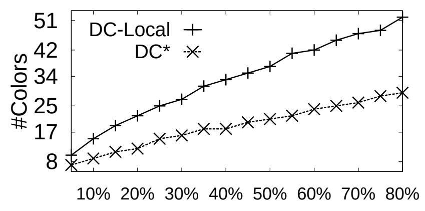
(d) *Trec*



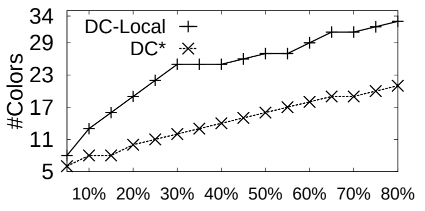
(e) *AS*



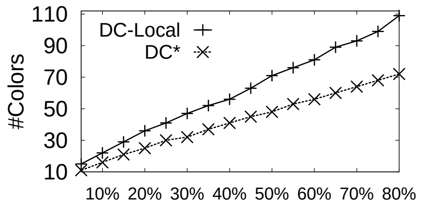
(f) *Epinion*



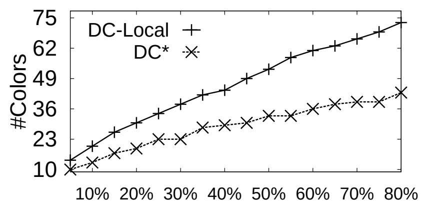
(g) *Baidu*



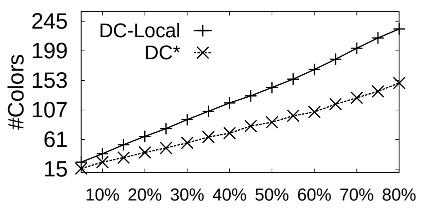
(h) *LastFM*



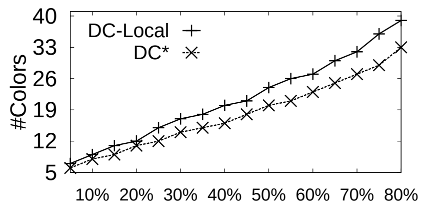
(i) *WikiTalk*



(j) *WikiEnglish*

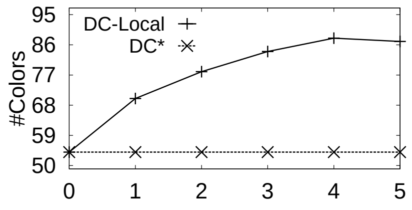


(k) *PL0*

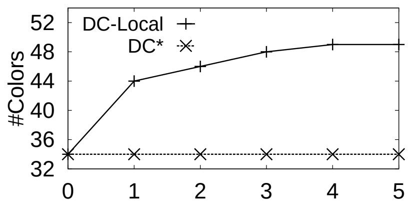


(l) *SSCA0*

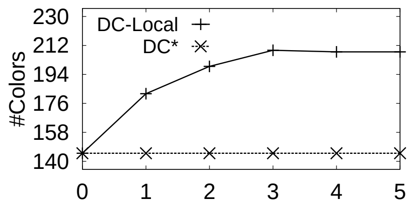
Figure 6.1: Coloring Quality



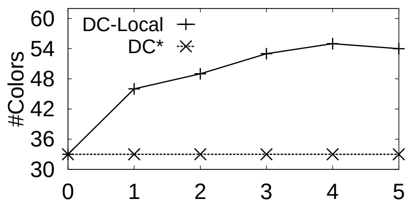
(a) *MoiveLens*



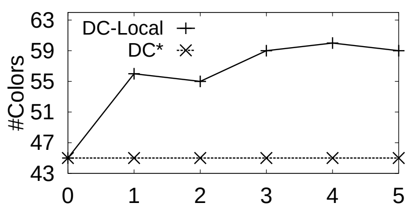
(b) *Libimseti*



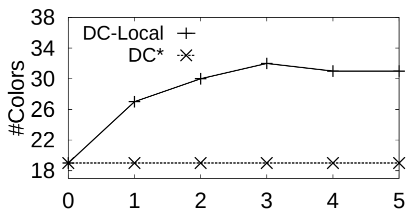
(c) *Flickr*



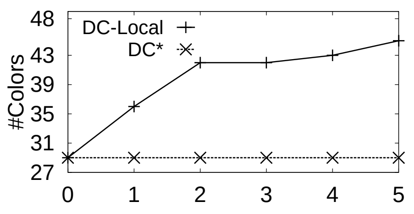
(d) *Trec*



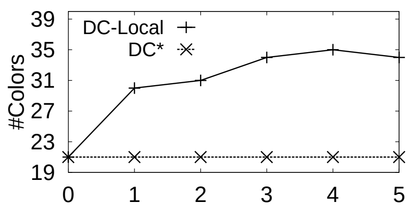
(e) *AS*



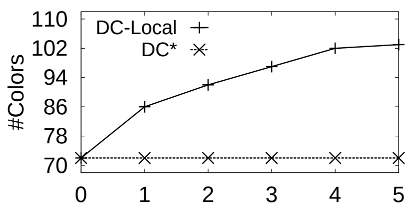
(f) *Epinion*



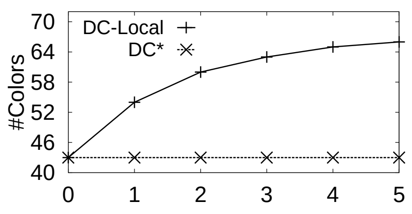
(g) *Baidu*



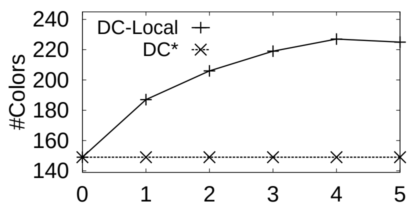
(h) *LastFM*



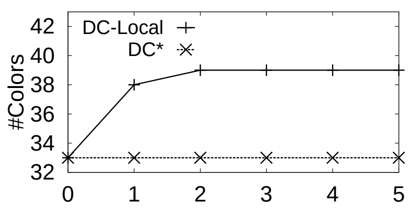
(i) *WikiTalk*



(j) *WikiEnglish*



(k) *PL0*



(l) *SSCA0*

Figure 6.2: Coloring Consistency

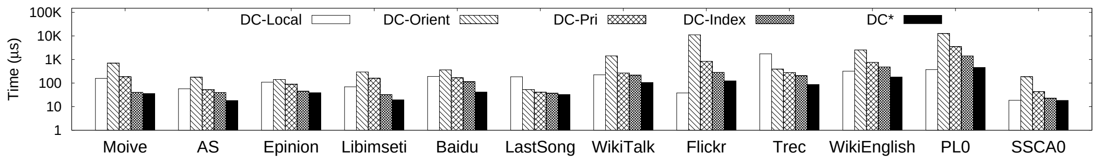


Figure 6.3: Average Processing Time

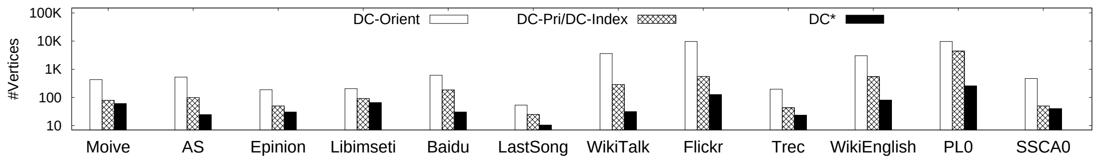


Figure 6.4: Average Number of Vertices Pushed in q

information alone and the local exploration leads to a bad coloring comparing to our approach when the graph is continuously updated. The reason for the stable movement after several updating procedures is that when the number of colors exceeds the optimal solution too much, we can easily find a coloring with such number of colors.

Exp-3: Processing Time for Each Update. In this experiment, we evaluate the efficiency of the five algorithms on all datasets. We randomly extract 20% of the edges from each graph as the edge pool and take the remaining part as the initial graph. We insert the edges from the pool into the initial graph and delete these edges randomly and record the processing time for each update. We color the initial graph by Global at the beginning and finish the experiment when each edge in the pool has been inserted and deleted once. The average processing time for each update are shown in Fig. 6.3.

As Fig. 6.3 shows, among our proposed algorithms, DC-Orient consumes the most time while DC-Pri only performs better than DC-Orient on all datasets. This is because the priority queue used in DC-Pri can reduce the unnecessary color reassignment caused by the out-of-order NC problem in DC-Orient. DC-Index further reduces the average processing time comparing to DC-Pri on all datasets. This is because using DINC-Index can avoid exploring the neighbors of type-2 vertices. DC* has the least average processing time among our proposed algorithms and its average processing time for each update is < 1 *ms* on all datasets. This is a natural result of the combination of the two proposed early pruning strategies. For example, on *WikiTalk*, the average time of DC* is 104 μs while that of DC-Index is 216 μs. Comparing to DC-Local, DC* is more efficient on ten of the twelve datasets. For example, on *WikiEnglish*, the average processing time of DC* is 129 μs while that of DC-Local is 318 μs. This is consistent with our analysis. Therefore, DC* is very efficient comparing to the other algorithms.

Exp-4: Number of Vertices pushed in q. In this experiment, we compare the average number of vertices pushed in q of our proposed algorithms for each update. We conduct the experiment similarly as Exp-3 and the results are shown in Fig. 6.4. As shown in Fig. 6.4, DC-Pri and DC-Index have the same average number of vertices pushed in q.

From Fig. 6.4, we can see: 1) for our proposed algorithms, the average number of vertices pushed in q is small comparing to $|V|$ of the graph. For example, on *WikiEnglish*, the average number of DC-Orient, DC-Pri/DC-Index and DC* are 3035, 545 and 81 respectively while $|V|$ of *WikiEnglish* is $18,268,992$. The reason for the small average number of vertices pushed in q is that we adopt the incremental computation strategy and we can achieve the global coloring quality without processing all the vertices in the graph. 2) the average number of DC-Pri/DC-Index is smaller than that of DC-Orient, and DC* has the least average number among these algorithms. For

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 6.2: φ for each update

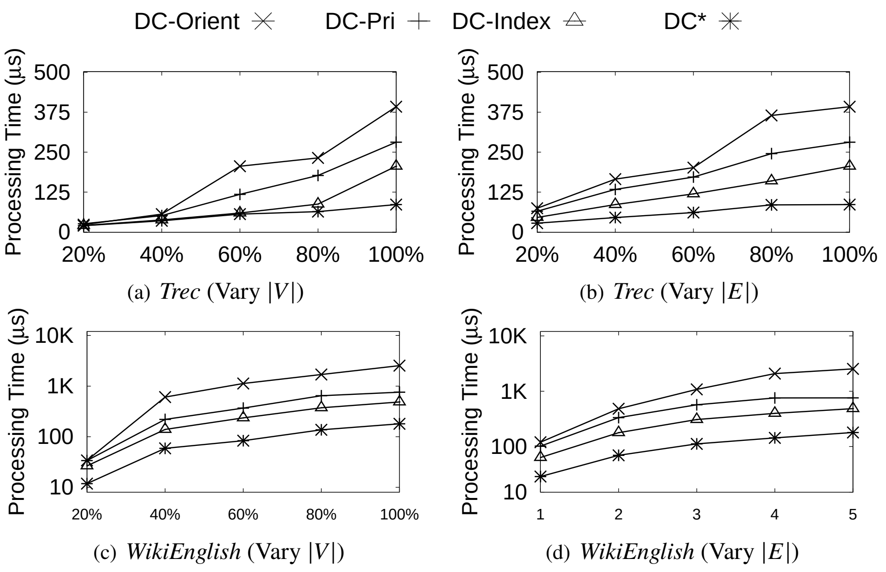


Figure 6.5: Scalability

example, the average number of DC-Orient, DC-Pri/DC-Index and DC* on *Baidu* are 613, 186 and 30 respectively. The reason that DC-Pri/DC-Index has a smaller average number than DC-Orient is that DC-Pri/DC-Index can avoid the out-of-order notification problem due to the priority queue. And because the notification pruning can further reduce the unnecessary notifications, the average number of DC* is smaller than that of DC-Pri/DC-Index.

Exp-5: φ for each update. Table 6.2 shows the φ (average number of vertices whose colors are changed) when an edge is inserted/deleted for the ten datasets used in our experiment. To compute φ, we insert and delete 10000 edges in a random manner for each dataset. And when an edge is inserted/deleted, we compute the new coloring by the Global algorithm and record the number of vertices whose colors are changed.

As Table 6.2 shows, when an edge is inserted/deleted, φ is very small comparing 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 ten datasets is 40.43 and the average φ for these ten datasets is only 11.4. Therefore, the number of vertices whose colors are changed is very small in practice, which confirms our observation in Section 1.

Exp-6: 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. 6.5.

As shown in Fig. 6.5 (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 reassigned their colors when an edge is inserted or deleted. Thus, the average processing time increases as $|V|$ increases. Of our proposed algorithms,

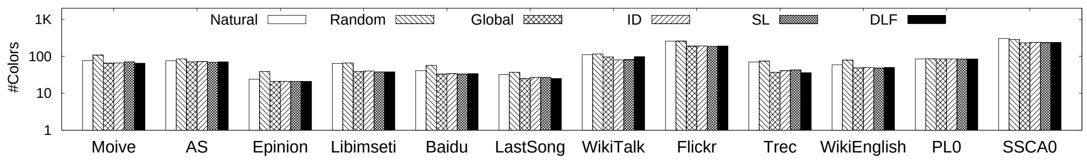


Figure 6.6: Static Graph Coloring Algorithms Comparison

DC* performs the best on all datasets. This is the result of the combination of the proposed optimization strategies. On all the datasets, the average processing time of DC* increases stably when $|V|$ increases while other algorithms may increase sharply. For example, on *Trec* (Fig. 6.5(a)), the average processing time of DC-Index has a sharp increment when we vary $|V|$ from 80% to 100% while DC* still increases stably. Thus, DC* has a good scalability. In Fig. 6.5 (b) and (d), when we vary $|E|$, we can find a similar trend as varying $|V|$.

Exp-7: Static Graph Coloring Algorithms Comparison. As our algorithms are built on Global, in this experiment, we compare the number of colors used by Global with other static graph coloring algorithms provided by ColPack on our experimental datasets to show the effectiveness of Global. ColPack is an open-sourced static graph coloring library (`https://github.com/CSCsw/ColPack/`) and is widely used in the literature [19]. Besides Global, it provides five other static graph coloring algorithms. All the provided algorithms share the same coloring framework which iterates over the vertices and assigns each vertex the smallest color not assigned to a neighbor, but the coloring orders of the algorithms are different. Specifically,

- Natural: it colors vertices in the order they appear in the input graph presentation.
- Random: it colors vertices in a uniformly random order.
- Global: it colors vertices in order of non-increasing degree (Algorithm 2 in Section 4).
- ID: it iteratively colors an uncolored vertex with the largest number of *colored* neighbors.
- SL: it colors the vertices in the order induced by first removing the lowest-degree vertices from the graph, then recursively coloring the resulting graph, and finally coloring the removed vertices.
- DLF: it iteratively colors an uncolored vertex with the largest number of *uncolored* neighbors. Note that DLF chooses the next uncolored vertex with the largest number of *uncolored* neighbors, while ID choose the vertex with the largest number of *colored* neighbors.

The experimental result is shown in Fig. 6.6.

In Fig. 6.6, Random uses the most number of colors on all datasets and Natural use less colors than Random but more colors than the remaining algorithms. For the remaining four algorithms, although the effectiveness of them are various on different datasets, the number of colors used by them are very close. For example, the number of colors used by Global, ID, SL and DLF on *Flickr* are 188, 191, 187 and 190, respectively. The experimental result is consistent with the result in [19]. Therefore, Global is competitive comparing with other state-of-the-art static graph coloring algorithms.

7 Related Work

Graph coloring is a fundamental problem in graph theory. Finding an optimal coloring for a graph is NP-complete in general [18]. [48] 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. Although some exact algorithms have been devised for this problem, such algorithms can only handle very small graphs [33, 41, 23, 20]. Since the exact algorithms do not work well in practice, many algorithms resorting to heuristics are proposed in the literature [42, 25, 37, 17, 36, 13, 24, 43, 44, 35]. Among them, Global [42] is the most popular due to its high efficiency in handling large graphs and high graph coloring quality in practice [31, 45, 2, 7]. A comprehensive survey on the graph coloring algorithms for static graphs can be found in [16].

There exist several studies on dynamic graph coloring problem in the literature. [11] studies the dynamic graph coloring problem on trees and product graphs and proves various dynamic chromatic number bounds on these types of graphs. [15] 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 [38], which is introduced in Section 3. Since DC-Local is the only dynamic graph coloring algorithm in the literature which has competitive effectiveness and efficiency, we choose it as a yardstick in our experiment.

The online graph coloring problem is closely related to our problem. Online graph coloring problem assumes that the vertices are given one by one (with their corresponding edges) and a color is assigned to the current vertex before the next vertex is colored. Once a color is assigned to a vertex, changes are not allowed. [22] proves that the lower bound of the performance ratio of any online graph coloring algorithm is $\Omega(n/\log^2 n)$. [21] proposes an algorithm with such lower bound. [32, 34, 10] mainly focus on coloring problem upon variants of online models and special classes of graphs.

Note that the dynamic coloring of a graph problem, which has a very similar name as our problem, is also investigated in the literature, such as [4, 6]. However, a dynamic coloring of a graph G is a proper coloring such that, for every vertex $v \in V(G)$ with degree at least 2, the neighbors of v receive at least 2 colors, which is totally different from dynamic graph coloring studied in this paper.

8 Conclusion

In this paper, we study the dynamic graph coloring problem. As the existing method is unable to maintain a high quality graph coloring, we aim to design a method that achieves the same coloring quality as one of the best static graph coloring algorithms while updating the coloring efficiently for dynamic graphs. We propose a color-propagation based algorithm on the oriented coloring graph to bound the explored vertices within the 2-hop neighbors of those vertices whose colors are changed in each graph update. We further improve our algorithm by devising a novel dynamic in-neighbor colors index and some pruning rules. The experimental results demonstrate the high effectiveness and efficiency of our approach.

Bibliography

[1] I. Abfalter. Nucleic acid sequence design as a graph colouring problem. *Ph.D Thesis*, 2005.

[2] A. Aboulnaga, J. Xiang, and C. Guo. Scalable maximum clique computation using mapreduce. In *Proc. of ICDE'13*, pages 74–85, 2013.

[3] C. Aggarwal and K. Subbian. Evolutionary network analysis: A survey. *ACM Computing Surveys (CSUR)*, 47(1):10, 2014.

[4] S. Akbari, M. Ghanbari, and S. Jahanbekam. On the list dynamic coloring of graphs. *Discrete Applied Mathematics*, 157(14), 2009.

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

[6] M. Alishahi. On the dynamic coloring of graphs. *Discrete Applied Mathematics*, 159(2), 2011.

[7] 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.

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

[9] N. Barnier and P. Brisset. Graph coloring for air traffic flow management. *Annals of operations research*, 130(1-4), 2004.

[10] M. P. Bianchi, H.-J. Böckenhauer, J. Hromkovič, and L. Keller. Online coloring of bipartite graphs with and without advice. *Algorithmica*, 70(1), 2014.

[11] P. Borowiecki and E. Sidorowicz. Dynamic coloring of graphs. *Fundamenta Informaticae*, 114(2), 2012.

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

[13] M. Chams, A. Hertz, and D. De Werra. Some experiments with simulated annealing for coloring graphs. *European Journal of Operational Research*, 32(2), 1987.

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

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

[16] P. Galinier, J.-P. Hamiez, J.-K. Hao, and D. Porumbel. Recent advances in graph vertex coloring. In *Handbook of optimization*. Springer, 2013.

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

[18] 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.

[19] 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 (TOMS)*, 40(1):1, 2013.

[20] S. Gualandi and F. Malucelli. Exact solution of graph coloring problems via constraint programming and column generation. *INFORMS Journal on Computing*, 24(1), 2012.

[21] M. M. Halldórsson. Parallel and on-line graph coloring. *Journal of Algorithms*, 23(2), 1997.

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

[23] P. Hansen, M. Labbé, and D. Schindl. Set covering and packing formulations of graph coloring: algorithms and first polyhedral results. *Discrete Optimization*, 6(2), 2009.

[24] J.-K. Hao and Q. Wu. Improving the extraction and expansion method for large graph coloring. *Discrete Applied Mathematics*, 160(16), 2012.

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

[26] 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. ACM, 2014.

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

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

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

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

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

[32] L. Ouerfelli and H. Bouziri. Greedy algorithms for dynamic graph coloring. In *Proceedings of CCCA*, 2011.

[33] J. Peemöller. A correction to brelaz's modification of brown's coloring algorithm. *Communications of the ACM*, 26(8), 1983.

[34] S. V. Pemmaraju, R. Raman, and K. Varadarajan. Max-coloring and online coloring with bandwidths on interval graphs. *ACM Transactions on Algorithms*, 7(3), 2011.

[35] 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*, 2016.

[36] 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), 2010.

[37] D. C. Porumbel, J.-K. Hao, and P. Kuntz. A search space cartography for guiding graph coloring heuristics. *Computers & Operations Research*, 37(4), 2010.

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

[39] J. Riihijärvi, M. Petrova, and P. Mähönen. Frequency allocation for wlans using graph colouring techniques. In *WONS*, volume 5, pages 216–222, 2005.

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

[41] E. Sewell. An improved algorithm for exact graph coloring. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 26, 1996.

[42] 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.

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

[44] Q. Wu and J.-K. Hao. An extraction and expansion approach for graph coloring. *Asia-Pacific Journal of Operational Research*, 30(05), 2013.

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

[46] A. Zaki, M. Attia, D. Hegazy, and S. Amin. Comprehensive survey on dynamic graph models. *International Journal OF ACSA*, 7(2):573–582, 2016.

[47] A. D. Zhu, W. Lin, S. Wang, and X. Xiao. Reachability queries on large dynamic graphs: a total order approach. In *Proceedings of SIGMOD*, pages 1323–1334. ACM, 2014.

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