Front. Comput. Sci., 2021, 15(4): 154610
https://doi.org/10.1007/s11704-020-9205-y

RESEARCH ARTICLE

# VColor*: a practical approach for coloring large graphs

**Yun PENG[1], Xin LIN (✉)[2], Byron CHOI[1], Bingsheng HE[3]**

1 Department of Computer Science, Hong Kong Baptist University, Hong Kong 999077, China
2 School of Computer Science and Technology, East China Normal University, Shanghai 200062, China
3 School of Computing, National University of Singapore, Singapore 119077, Singapore



**Abstract** Graph coloring has a wide range of real world applications, such as in the operations research, communication network, computational biology and compiler optimization fields. In our recent work [1], we propose a divide-and-conquer approach for graph coloring, called `VColor`. Such an approach has three *generic subroutines*. (i) *Graph partition subroutine*: `VColor` partitions a graph $G$ into a vertex cut partition (VP), which comprises a vertex cut component (VCC) and small non-overlapping connected components (CCs). (ii) *Component coloring subroutine*: `VColor` colors the VCC and the CCs by efficient algorithms. (iii) *Color combination subroutine*: `VColor` combines the local colors by exploiting the maximum matchings of color combination bigraphs (CCBs). `VColor` has revealed some major bottlenecks of efficiency in these subroutines. Therefore, in this paper, we propose `VColor*`, an approach which addresses these efficiency bottlenecks without using more colors both theoretically and experimentally. The technical novelties of this paper are the following. (i) We propose the *augmented VP* to index the crossing edges of the VCC and the CCs and propose an optimized CCB construction algorithm. (ii) For sparse CCs, we propose using a greedy coloring algorithm that is of polynomial time complexity in the worst case, while preserving the approximation ratio. (iii) We propose a distributed graph coloring algorithm. Our extensive experimental evaluation on real-world graphs confirms the efficiency of `VColor*`. In particular, `VColor*` is 20X and 50X faster than `VColor` and uses the same number of colors with `VColor` on the `Pokec` and `PA` datasets, respectively. `VColor*` also significantly outperforms the state-of-the-art graph coloring methods.

**Keywords** graph coloring, approximation algorithm, distributed algorithm

## 1 Introduction

The *graph coloring problem* is to color the vertices of a graph using the fewest colors such that no two adjacent vertices are having the same color. For quick reference, we present two example graphs and their possible colorings in Fig. 1.

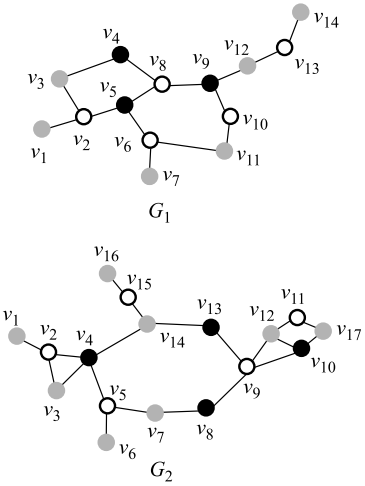


**Fig. 1** Example graphs and their possible colorings

The importance of graph coloring has long been recognized in the literature of operations research, communication network, computational biology and compiler optimization, among others:

- *Nucleic acid sequence design*. Given a set of nucleic acids, a dependency graph [2] is a graph, where each vertex is a nucleotide and two vertices have an edge if and only if the two nucleotides form a base pair in at least one of the nucleic acids. A coloring of the dependency graph is a nucleic acid sequence that is compatible with the set of nucleic acids.
- *Frequency assignment*. A cellular phone network is modeled as a graph, where a vertex is a base station and two vertices are neighbors iff the two base stations are in communication range. When assigning frequencies to the base stations, the neighbors that are close to each other need to be assigned to different frequencies to avoid interference. A frequency assignment is exactly a graph coloring of the network [3].
- *Compiler optimization*. The register allocators of almost all modern production compilers are based on graph coloring [4]. Specifically, given a set of registers and values, one may construct an interference graph, where a vertex is the live range of values, and an edge indicates that the two live ranges have overlaps and interfere with each other.

Received June 5, 2019; accepted January 6, 2020

E-mail: xlin@cs.ecnu.edu.cn

Register allocation is equivalent to coloring the interference graph.

- *Scheduling*. Assume that we have to schedule a set of interferencing jobs (*e.g.*, scheduling aircrafts to flights). We can construct a conflict graph, where the vertices are jobs, and two vertices have an edge iff the corresponding jobs cannot be executed at the same time. Let colors denote available time slots and each job needs a time slot. The coloring of the conflict graph with a minimum number of colors is the schedule with the smallest time span [5].
- *Community detection*. Community detection on a large social network that may have millions of nodes is challenging. Graph coloring is often used to compute a set of seed vertices, from which high quality overlapping communities of the social network can be constructed [6].
- *Clique computation*. Graph coloring also plays a crucial role in computing cliques of a large grpah [7].

Graph coloring has known to be an NP-hard problem and notable efforts have been spent on establishing its heuristic or approximation algorithms (e.g., [8–13]). Halldrsson [9] have proposed `SampleIS`, which has currently the best-known approximation ratio $|G|(\log\log|G|)^2/\log^3|G|$, where $G$ is the input graph and $|G|$ is the number of vertices in $G$. However, its time complexity is $O(|G|^3)$. It is hence impractical to cope with the scale of real graphs nowadays. For example, in our preliminary experiments running on a commodity machine, `SampleIS` *took 2,381 seconds to color a Latin Square graph* `LS` *with just 0.9K vertices, and did not finish after running for one week to color a road network (having 260K vertices).*

In our recent work [1], we propose a vertex-cut based coloring method (`VColor`). `VColor` is a novel *divide-and-conquer* framework. Such a framework has three generic subroutines, as illustrated in Fig. 2.

However, the three subroutines reveal three efficiency bottlenecks of the divide-and-conquer framework (marked by bold in Fig. 2). *Firstly*, the construction of the Color Combination Bigraph (CCB) in the subroutine-(iii) is inefficient as the edges crossing the VCC and the CCs are scanned redundantly for many times. Suppose $G$ is partitioned to a VCC and $k$ CCs and let $\mathcal{I}_{VCC}$ and $\mathcal{I}_{CC_i}$ denote the colors (i.e., Independent Sets (ISs)) of the VCC and the $CC_i$, respectively. Constructing the CCBs for the VCC and $k$ CCs takes time $O(|\mathcal{I}_{VCC}||E(G)| + \sum_{i=1}^{k}|\mathcal{I}_{CC_i}||VCC|)$. Our experiments on `Pokec` shows that the construction of the CCB for all the 54,345 CCs can take more than 90% of the total runtime of `VColor`. *Secondly*, `VColor` enumerates all Maximal ISs (MISs) of each CC to color the CC in the subroutine-(ii). It is inefficient as the time complexity of enumerating all MISs of a CC is $O(3^{|CC|/3})$. *Thirdly*, the subroutine-(ii) colors the CCs sequentially and the subroutine-(iii) computes the CCBs sequentially. However, each CC can be colored independently from other CCs and the construction of each CCB can be computed independently from other CCBs. For example, on `Pokec`, at least half of the CCs can be colored independently. Therefore, the subroutine-(ii) and subroutine-(iii) can be processed in a distributed manner.

In this paper, we propose `VColor*`, which significantly optimizes `VColor`. *Firstly*, we index the edges crossing the VCC and the CCs, such that the crossing edges only need to be scanned for one time. Given the colors of the VCC and the CC, we first construct a complete CCB $X$. Then, for each crossing edge $(u, u')$, we delete the edge $(I, I')$ from $X$, where $u \in I$ and $u' \in I'$. In this way, determining the edges of $X$ just needs one scan on the crossing edges of the VCC and the CC. The time complexity of the optimized CCB construction is reduced to $O(|V(G)| + |E_{cross}| + \sum_{i=1}^{k}(|\mathcal{I}_{CC_i}||\mathcal{I}_{VCC}|))$, where $E_{cross}$ is the set of crossing edges. Our experiments on `Pokec` and `PA` show that the construction of $X$ can be 10X faster.

*Secondly*, we propose a theorem that when a CC is sparse (i.e., $\log|CC| \geqslant \Delta$ and $\Delta$ is the largest vertex degree of the CC), we can color the CC using `Greedy` [14]. The time complexity is reduced from $O(3^{|CC|/3})$ to polynomial of $|CC|$ and the approximation ratio is no worse than that of the MIS enumeration based method. Based on the theorem, a hybrid algorithm is proposed to optimize the coloring of the CCs. Our experiments show that this optimization can reduce running time by ∼10%.

*Thirdly*, we propose a distributed graph coloring algorithm. The master holds the color of the VCC. The slaves color the CCs, and compute the CCBs and MMs. The colors of the CCs and the Maximum Matchings (MMs) are returned to the master.

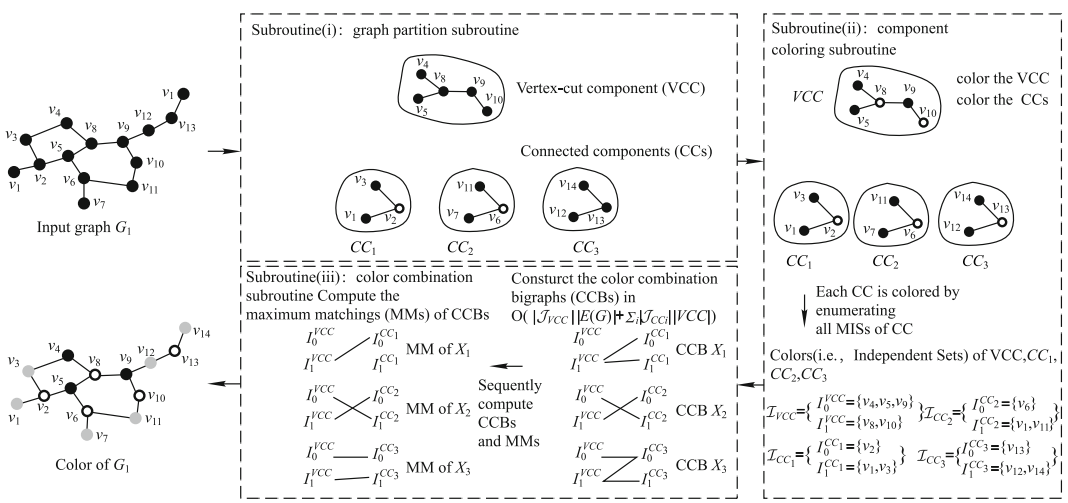


**Fig. 2** Overview of three subroutines (the bold sentences show the bottlenecks)

The master combines the colors of the VCC and the CCs using the MMs. Our algorithm significantly outperforms the state-of-the-art distributed graph coloring method KW [15]. Our experiments on Pokec and PA show that given roughly the same coloring time, we use 10000X fewer colors than KW.

The contributions of this paper are as follows.

- We review the recent vertex-cut based graph coloring approach VColor [1]. We analyze the three generic subroutines and present three efficiency bottlenecks of VColor.
- For the efficiency bottleneck of the color combination subroutine, we propose an index (augmentation of VP), which can optimize the CCB construction.
- For the efficiency bottleneck of the component coloring subroutine, we propose using the greedy algorithm to color the sparse CCs. We analyze that the time complexity is reduced and the approximation ratio is preserved.
- We propose a distributed graph coloring algorithm. It significantly outperforms the state-of-the-art distributed methods KW and JP.
- Our experiments verify the effectiveness and efficiency of our techniques on real-world graphs that have up to millions of nodes. In particular, VColor* is 20X and 50X faster than VColor and uses the same number of colors with VColor on the Pokec and PA datasets, respectively.

**Organizations** The rest of this paper is organized as follows. Section 2 provides the background of this paper. Section 3 reviews VColor. Section 4 presents the techniques of VColor*. The experiment results are reported in Section 5. Section 6 discusses the related works, and Section 7 concludes this paper.

## 2 Preliminaries and problem definition

We start by recalling some relevant notations for graph coloring. This paper studies *undirected graphs*, or simply called *graphs*. A graph is denoted as $G = (V, E)$, where $V(G)$ and $E(G)$ are the vertex set and the edge set of $G$, respectively. $|G|$ denotes the size of $G$ and $|G| = |V(G)|$. $N(v)$ and $N(S)$ denote the neighbors of $v \in V(G)$ and $S \subseteq V(G)$, respectively. $\overline{N}(v)$ and $\overline{N}(S)$ denote the non-neighbors of $v$ and $S$, respectively. $\Delta$ denotes the largest degree of vertices in $G$. A *vertex cut* of $G$ is a set of vertices of $G$ whose removal makes $G$ disconnected. An *independent set* (IS) $I$ of $G$ is a subset of $V(G)$, such that $\forall u, v \in I$, $(u, v) \notin E(G)$. A *maximal independent set* (MIS) $M$ of $G$ is an IS, such that $M \cup \{v\}$ is not an IS, for any $v \in V(G) \backslash M$. $\mathcal{M}(G)$ denotes the set of all MISs of $G$. An *independent set partition* $\mathcal{I}$ of $G$ is a set of non-empty subsets of $V(G)$, where $\forall I \in \mathcal{I}$ is an IS of $G$, $\forall I, I' \in \mathcal{I}, I \cap I' = \emptyset$ and $\cup_{I \in \mathcal{I}} I = V(G)$. The size of an IS partition $\mathcal{I}$ is the number of ISs in it. An IS partition $\mathcal{I}$ is *minimal*, if $\forall I_1, I_2 \in \mathcal{I}$, $(\mathcal{I} \backslash \{I_1, I_2\}) \cup (I_1 \cup I_2)$ is not an IS partition of $G$.

**Definition 1** A coloring of $G$ is an assignment of a color to each vertex of $G$ such that no two neighboring vertices are assigned the same color.

If a graph $G$ can be colored using $\alpha$ colors, $G$ is called *$\alpha$-colorable*. The minimum value of $\alpha$ is called the *chromatic number* of $G$, denoted by $\chi_G$. The set of vertices assigned with the same color is called a *color class*.

**Proposition 1** An $\alpha$-coloring of $G$ is equivalent to an IS partition of $G$ of size $\alpha$, where each IS is a color class.

**Problem definition** *Given a graph G, color G with the fewest colors.*

## 3 Vertex-cut Based Graph Coloring (VColor)

In this section, we briefly review the VColor framework proposed in [1]. This facilitates our discussions on the efficiency bottleneck of such a framework. The frequently used symbols of VColor are summarized in Table 1. The VColor framework unleashes the *divide-and-conquer* approach to graph coloring and has three main subroutines as follows.

**(i) Graph partition subroutine** We partition the input graph $G$ into a set of connected components (CCs) of a small size $s$ by removing a vertex cut component (VCC). Such a partition is called the vertex cut partition (VP). The rationale of VP is as follows.

- Since a CC is small, we can afford a method of exponential time complexity to color a CC to provide a better approximation ratio than SampleIS;
- If the CCs totally account for the majority of $G$, we can obtain a better coloring of $G$ than SampleIS;
- There is no crossing edge between the CCs. Therefore, we can color the CCs independently and the colors of the CCs can be efficiently combined.

The VP is defined as follows.

**Definition 2** Given a graph $G$ and a parameter $s$, a *Vertex Cut Partition* (VP) of $G$ is a graph partition $\mathcal{P}$=$\{CC_1, CC_2, ..., CC_k, VCC\}$, where $VCC$ is the vertex cut component, removing which leads to connected components $\{CC_1, ..., CC_k\}$ of size $s$. $V(G) = (\cup_{i=1}^{k} V(CC_i)) \cup V(VCC)$.

Note that there may often exist $CC$s that are smaller than $s$, but such cases are omitted for the simplicity of presentation and analysis. We use $VCC(\mathcal{P})$ to denote the $VCC$ of $\mathcal{P}$ and $CC(\mathcal{P})$ to denote the $CC$'s $\{CC_1, ..., CC_k\}$ of $\mathcal{P}$.

**Example 1** Suppose the size of CC is $s = 3$. Figure 3(a) presents the VP of $G_1$ of Fig. 1.

**(ii) Component coloring subroutine** We color the VCC and each CC separately, as shown in Fig. 4. For the VCC, we simply adopt SampleIS (Line 01). For the CC, we propose an MIS enumeration based method colorCC_*by_MISE* (Lines 02–03).

Specifically, for each $CC_i$ in the VP $\mathcal{P}$ of $G$, colorCC_*by_MISE* first enumerates all the MISs of $CC_i$, using the MIS enu-

**Table 1** Frequently used symbols of VColor

| | |
|---|---|
| $\|G\|$ | The size of $G$, given by the number of vertices in $G$ |
| $\mathcal{P}, \mathcal{P}_G$ | The vertex cut partition (VP) of $G$ |
| $VCC(\mathcal{P})$ | The vertex cut component (VCC) in $\mathcal{P}$ |
| $CC(\mathcal{P})$ | The set of connected components (CCs) in $\mathcal{P}$ |
| $s$ | The size of a connected component (CC) |
| $\Delta, \Delta_G$ | The largest degree of the vertices of $G$ |
| $\mathcal{I}_{VCC}, \mathcal{I}_{CC_i}$ | The color of $VCC$ and $CC_i$, resp. |
| $X$ | The color combination bigraph (CCB) |

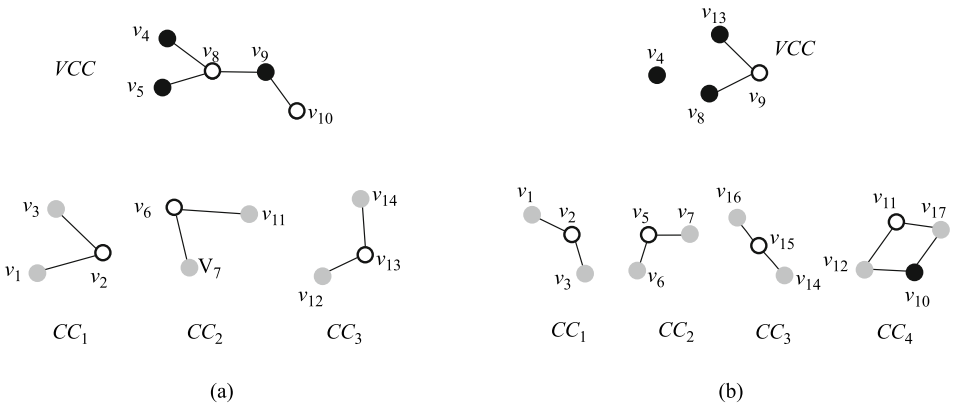


**Fig. 3** Examples of vertex cut partitions of $G_1$ and $G_2$. (a) VP $P_1$ of $G_1$; (b) VP $P_2$ of $G_2$

```
Procedure Color
Input: A graph G, a VP 𝒫 of G
Output: A minimal IS partition (i.e. color) of G

01 ℐ_VCC = SampleIS(VCC) //color the VCC, subroutine-(ii.A)
02 for each CC_i in 𝒫
03    ℐ_CC_i = colorCC_by_MISE(CC_i) // subroutine-(ii.B)
04 return comb(ℐ_VCC, ℐ_CC_1, ℐ_CC_2, ..., ℐ_CC_k, G)

function colorCC_by_MISE(CC_i)
05 ℳ(CC_i) = MISEnum(CC_i) //enumerate all MISs of CC_i
06 return ℐ_CC_i = ISPartition(ℳ(CC_i), CC_i)

function ISPartition(ℳ, G) // subroutine-(i)
07 ℐ = ∅
08 while |∪_{I∈ℐ} I| < |G|
09    M = arg max_{M∈ℳ} |∪_{I∈ℐ} I ∪ {M}| − |∪_{I∈ℐ} I|
10    I = M − ∪_{I∈ℐ} I
11    add I to ℐ
12 return ℐ

function comb(ℐ_VCC, ℐ_CC_1, ℐ_CC_2, ..., ℐ_CC_k, G) //subroutine-(iii)
13 for each i = 1 to k
14    construct an empty bigraph X //initialize the CCB
15    for each I ∈ ℐ_VCC, insert a vertex to X
16    for each J ∈ ℐ_CC_i, insert a vertex to X
17    for each I ∈ ℐ_VCC and I′ ∈ ℐ_CC_i
18       if N(I) ∩ I′ = ∅
19          insert an edge (I, I′) to X
20    compute a maximum matching M of X
21    for each I′ ∈ ℐ_CC_i
22       if ∃I ∈ ℐ_VCC s.t. (I, I′) ∈ M
23          I = I ∪ I′
24       else add I′ to ℐ_VCC
25 return ℐ_VCC
```

**Fig. 4** Procedure `Color`

meration algorithms (e.g., [16,17]). Then, `colorCC_by_MISE` computes a minimal IS partition of $CC_i$ by `ISPartition` to cover $CC_i$. `ISPartition` is based on the heuristic of `SetCover` (Lines 07-12).

*Efficiency bottleneck analysis.* Firstly, `VColor` uses the MIS enumeration (MISE) based method on each CC. It can take an exponential time of the size of the CC to provide a better approximation ratio than `SampleIS`. But, the structure of the CC has not been studied. In particular, we may not use the MISE based method on very sparse CCs. Secondly, `VColor` colors the CCs sequentially. However, the CCs can be colored independently in a distributed manner.

**(iii) Color combination subroutine** Since the colors of the CCs can be combined easily as there is no edge of $G$ crossing the CCs, we just need to study how to combine the colors of the VCC and the CCs.

The main idea is that for an IS $I$ in $VCC$ and an IS $I'$ of $CC_i$, if there is no edge of $G$ crossing $I$ and $I'$, $I$ and $I'$ can be combined. We first define the color combination bigraph as follows. Given the coloring of $VCC$ and that of $CC_i$, we construct a color combination bigraph (CCB) $X = (\mathcal{I}_{VCC} \cup \mathcal{I}_{CC_i}, E_X)$, where each IS in $\mathcal{I}_{VCC}$ and $\mathcal{I}_{CC_i}$ is a vertex of $X$. $\mathcal{I}_{VCC}$ is one part of $X$ and $\mathcal{I}_{CC_i}$ is the other part of $X$. For $I \in \mathcal{I}_{VCC}$ and $I' \in \mathcal{I}_{CC_i}$, $(I, I') \in E_X$ iff $G$ has no edge $(v, v')$ satisfying $v \in I, v' \in I'$. Then, computing the optimum combination of the colorings of $VCC$ and $CC_i$ is equivalent to computing the maximum matching (MM) of $X$.

The function `comb` of Fig. 4 constructs the CCB for $VCC$ and each $CC_i$ and uses the MM of the CCB to combine the colors of the $VCC$ and $CC_i$. The final combination result is an optimal color of $G$.

*Efficiency bottleneck analysis.* Firstly, the construction algorithm of the CCB $X$ (Function `comb` Lines 14–19) can be inefficient due to the redundant scanning of the crossing edges of the VCC and CCs. Secondly, the CCBs and the MMs of the CCBs are computed sequentially. However, they can be computed independently in a distributed manner.

**Example 2** We illustrate the three subroutines by coloring the graph $G_1$ in Fig. 1.

We use the vertex cut partition shown in Fig. 3(a).

We color $VCC$ using `SampleIS` and obtain $\mathcal{I}_{VCC} = \{\{v_4, v_5, v_9\}, \{v_8, v_{10}\}\}$.

We color each CC using `colorCC_by_MISE` and obtain the following.

- $\mathcal{I}_{CC_1} = \{\{v_1, v_3\}, \{v_2\}\}$,
- $\mathcal{I}_{CC_2} = \{\{v_7, v_{11}\}, \{v_6\}\}$, and
- $\mathcal{I}_{CC_3} = \{\{v_{12}, v_{14}\}, \{v_{13}\}\}$.

We combine the colors of the VCC and each CC as follows.

- Initially, $\mathcal{I}_{VCC} = \{\{v_4, v_5, v_9\}, \{v_8, v_{10}\}\}$;
- After combining $\mathcal{I}_{CC_1}$, $\mathcal{I}_{VCC}$ becomes $\{\{v_4, v_5, v_9\}, \{v_2, v_8, v_{10}\}, \{v_1, v_3\}\}$;
- After combining $\mathcal{I}_{CC_2}$, $\mathcal{I}_{VCC}$ becomes $\{\{v_4, v_5, v_9\}, \{v_2, v_6, v_8, v_{10}\}, \{v_1, v_3, v_7, v_{11}\}\}$;

- Finally, $\mathcal{I}_{VCC} = \{\{v_4, v_5, v_9\}, \{v_2, v_6, v_8, v_{10}, v_{13}\}, \{v_1, v_3, v_7, v_{11}, v_{12}, v_{14}\}\}$ after combining $\mathcal{I}_{CC_3}$.

**Proposition 2** [1] Given a VP $\mathcal{P} = \{CC_1, ..., CC_k, VCC\}$ of a graph $G$, the approximation ratio of Procedure `Color` is $\log s + 1 + |VCC|(\log\log|VCC|)^2/\log^3|VCC|$.

We remark that the approximation ratio of Procedure `Color` is dependent on $|VCC|$ and $s$, and equals to that of `SampleIS` in the worst case.

**Proposition 3** Given a VP $\mathcal{P} = \{CC_1, ..., CC_k, VCC\}$ of a graph $G$, the time complexity of Procedure `Color` is $O(s^2 3^{s/3} + |VCC|^3 + kt_{CCB} + k\sqrt{2}(\Delta_G)^{2.5})$, where $\Delta_G$ is the largest vertex degree of $G$ and $t_{CCB}$ is the time to construct the CCB.

**The vertex cut partition construction** Proposition 3 presents that when $s$ is fixed, it is desirable to minimize the size of the $VCC$ of the VP of $G$. However, computing the optimum VP is an NP-hard problem.

**Theorem 1** Given a graph $G$ and a parameter $s$, it is NP-hard to construct a VP $\mathcal{P}$ of $G$ such that the size of $VCC$ in $\mathcal{P}$ is minimized.

**Proof** (Sketch) The problem is clearly in NP. We then prove that the minimum balanced $\alpha$-vertex separator (MBVS) problem, which is NP-hard [18], can be reduced to it. Specifically, given a MBVS instance with a graph $G$ and a value of $\alpha$, an instance of the minimum VCC problem can be constructed on $G$ and set $s{=}\alpha|V|$. Then, a VP with the minimum $VCC$ is a solution of MBVS. □

We hence propose a heuristic algorithm to compute a VP of $G$ with a minimal $VCC$. The algorithm is presented in Fig. 5. The main idea is that in each iteration we use the subgraphs of $G$ that have the *minimal* neighborhood as the $CC$s. Specifically, we use the logic of BFS on $G$ to explore a subgraph $S$ of size $s$ (Lines 03-07). To minimize the neighborhood $N(S, G)$ of $S$, in each iteration of BFS, we pick the vertex $v \in G$ that can minimize $N(v, G)\backslash S$. (Lines 04,06). $S$ is added to $\mathcal{P}$ and $N(S, G')$ is added to $VCC$ (Lines 08-09). $S$ and $N(S, G')$ are removed from $G'$ for the next iteration (Line 10).

**Example 3** Consider the graph $G_1$ in Fig. 1, we show how to compute the VP $\mathcal{P}_1$ shown in Fig. 3(a). Let $s = 3$. $CC_1$ is com

**Procedure** `VP_cons`
**Input**: A graph $G$, component size $s$
**Output**: A minimal vertex cut partition

01 $\mathcal{P} = \emptyset$, $VCC = \emptyset$, $G' = G$
02 **while** $|G'| > s$
03 $S = \emptyset$
04 add a vertex $v$, s.t. $N(v, G')$ is the smallest, to $S$
05 **while** $|S| < s$
06 pick $v$ from $N(S, G')$ s.t. $N(v, G')\backslash S$ is the smallest
07 add $v$ to $S$
08 $\mathcal{P} = \mathcal{P} \cup \{S\}$
09 $VCC = VCC \cup N(S, G')$
10 remove $S$ and $N(S, G')$ from $G'$
11 $\mathcal{P} = \mathcal{P} \cup \{G'\} \cup \{VCC\}$
12 **return** $\mathcal{P}$

**Fig. 5** Procedure `VP_cons`

puted as follows. We first add $v_1$ to $CC_1$ because it is one of the vertices of the smallest degree in $G_1$ and $VCC = \{v_2\}$. Since $|CC_1| < 3$, we need to add neighbors of $CC_1$ into $CC_1$. Since $v_1$ only has one neighbor $v_2$, we add $v_2$ to $CC_1$ and $VCC$ becomes $\{v_3, v_5\}$. We need to pick one more node in $N(CC_1)$ to $CC_1$. $v_2$ has two neighbors, $v_3$ and $v_5$, and we need to compare them. If we add $v_3$ to $CC_1$, $VCC = \{v_4, v_5\}$. If we add $v_5$ to $CC_1$, $VCC = \{v_3, v_6, v_8\}$, which is larger than that of $v_3$. Therefore, $v_3$ is better and we add $v_3$ to $CC_1$. $CC_1 = \{v_1, v_2, v_3\}$ and $VCC = \{v_4, v_5\}$ and the computation of $CC_1$ finishes. The same logic is applied to $G_1\backslash(CC_1 \cup VCC)$. Finally, we have $CC_2 = \{v_6, v_7, v_{11}\}$ and $CC_3 = \{v_{12}, v_{13}, v_{14}\}$ and $VCC = \{v_4, v_5, v_8, v_9\}$.

Note that if the VCC is still large, we can recursively partition the VCC and construct a VP Hierarchy (`VPH`) of $L$ levels. The details of `VPH` are presented in [1]. The optimal values of $s$ and $L$ can be decided by preliminary experiments on the certain graphs.

## 4 Optimized vertex-cut based graph coloring (`VColor*`)

In this section, we propose `VColor*` to optimize `VColor`. Figure 6 presents an overview of `VColor*`. `VColor*` follows the three subroutines of the divide-and-conquer framework of `VColor` yet addressing the major performance bottlenecks of the framework. Specifically, firstly, the construction of the bigraph $X$ (for computing MM) is optimized by indexing. Secondly, sparse CCs are colored by `Greedy` for efficiency. Thirdly, we propose a distributed graph coloring algorithm. The frequently used symbols of `VColor*` are summarized in Table 2.

### 4.1 Optimizing the construction of the color combination bigraph $X$

In subroutine-(iii), to combine the colors of a given $VCC$ and $CC_i$ of $G$, a natural approach is to construct a bigraph $X_i = (\mathcal{I}_{VCC} \cup \mathcal{I}_{CC_i}, E)$, where $\mathcal{I}_{VCC}$ is the set of independent sets

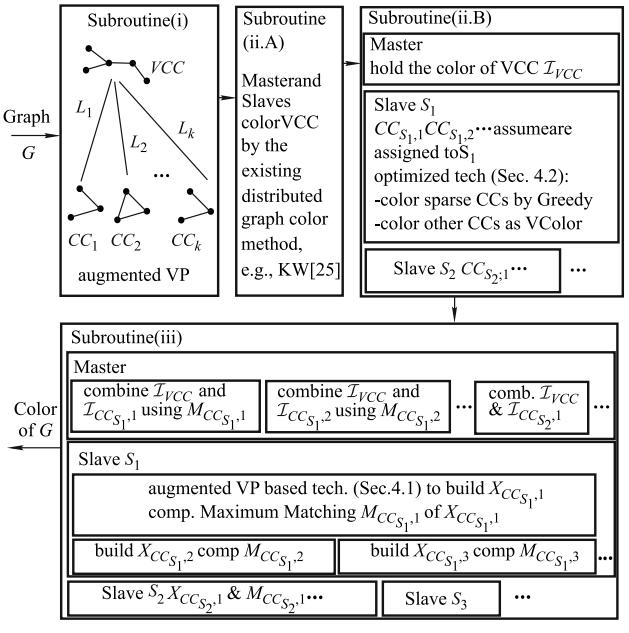


**Fig. 6** An overview of VColor*

**Table 2** Frequently used symbols of VColor*

| Symbol | Meaning |
|---|---|
| $\mathcal{P}^+$ | Augmented VP of $G$ |
| $L_i = label(VCC, CC_i)$ | The label between $VCC$ and $CC_i$ in $\mathcal{P}^+$ |
| $BI$ | The boundary information of $VCC$ |
| $m$ | The number of slaves |
| $S_j$ | The $j$-th slave |
| $M2S_j$ | The message from the master to $S_j$ |
| $S_j2M$ | The message from $S_j$ to master |
| $ms_{vcc}$ | The size of the messages for coloring the VCC |
| $t_{vcc}$ | The time to color the VCC |
| MM | Maximum matching |
| $\mathcal{I}, X, VCC, CC$ | Refer to Table 1 |

(ISs) of $VCC$, $\mathcal{I}_{CC_i}$ is the set of ISs of $CC_i$ and $X_i$ has an edge between two ISs in $E$ iff $G$ has edges crossing the two ISs. (Recall that all vertices in one IS have the same color and different ISs have different colors.) Then, the maximum matching (MM) of $X_i$ is computed. If two ISs are matched, they can be combined/merged to one IS and the number of colors used is reduced by one. However, we observe that the construction of the $X_i$'s of the $CC_i$'s (Lines 13–19 of comb of Fig. 4) dominates the overall coloring time.

The main reason is that when combining the colors of $VCC$ and $CC_i$, the subroutine requires to scan the crossing edges of $VCC$ and $CC_i$ many times. More specifically, for each IS $I \in \mathcal{I}_{VCC}$, the subroutine scans $I$ and $N(I')$ to check if $I \cap N(I') = \emptyset$ for each IS $I' \in \mathcal{I}_{CC_i}$. Scanning all the ISs of $VCC$ is to scan $VCC$. Scanning $N(I')$ for all ISs $I'$'s of $CC_i$ is to scan $CC_i$ and the crossing edges of $VCC$ and $CC_i$. Therefore, the subroutine requires to scan $CC_i$ and the crossing edges of $VCC$ and $CC_i$ for $|\mathcal{I}_{VCC}|$ times.

**Example 4** Consider the $VCC$ and $CC_2$ in Fig. 7. The ISs of $VCC$ and $CC_2$ are shown in Fig. 10(a). To construct the bigraph $X_2$ shown in Fig. 10(c), the crossing edges of $VCC$ and $CC_2$ need to be scanned for two passes by VColor (the function comb of Fig. 4).

Such repeated scanning on the crossing edges of $VCC$ and $CC_i$ (in order to determine if $I \cap N(I') \neq \emptyset$ for the $I$'s of $VCC$ and the $I'$'s of $CC_i$) results in a high time complexity, as follows.

**Proposition 4** Given a VP $\mathcal{P} = \{CC_1, CC_2, ..., CC_k, VCC\}$ of $G$, the total time to construct the bigraphs for $VCC$ and each $CC$ is $O(|\mathcal{I}_{VCC}||E(G)| + \sum_{i=1}^{k} |\mathcal{I}_{CC_i}||VCC|)$.

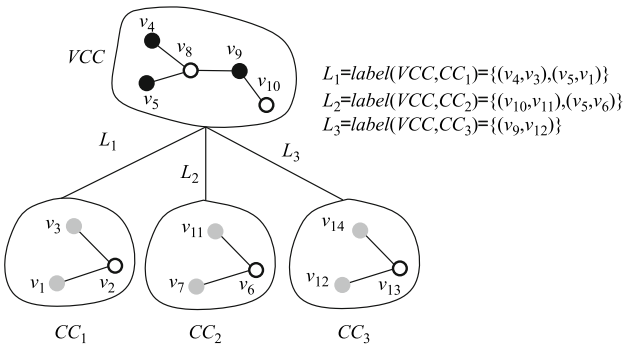


**Fig. 7** Augmented VP

**Proof** Let $\mathcal{I}_{CC_1}$, $\mathcal{I}_{CC_2}$, ..., $\mathcal{I}_{CC_k}$, $\mathcal{I}_{VCC}$ denote the colorings (i.e., the IS partitions) of $CC_1, CC_2, ..., CC_k, VCC$, respectively.

Consider $CC_i$. To construct the bigraph $X_i$ for $VCC$ and $CC_i$, comb examines $I \cap N(I')$ for each $I \in \mathcal{I}_{VCC}, I' \in \mathcal{I}_{CC_i}$. The time complexity is

$$O(\sum_{I' \in \mathcal{I}_{CC_i}} (\sum_{I \in \mathcal{I}_{VCC}} (|I| + |N(I')|)))$$
$$= O(\sum_{I' \in \mathcal{I}_{CC_i}} (|VCC| + |\mathcal{I}_{VCC}||N(I')|)).$$

The total time for all CCs is

$$O(\sum_{i=1}^{k} \sum_{I' \in \mathcal{I}_{CC_i}} (|VCC| + |\mathcal{I}_{VCC}||N(I')|))$$
$$= O(|\mathcal{I}_{VCC}||E(G)| + \sum_{i=1}^{k} \sum_{I' \in \mathcal{I}_{CC_i}} |VCC|)$$
$$= O(|\mathcal{I}_{VCC}||E(G)| + \sum_{i=1}^{k} |\mathcal{I}_{CC_i}||VCC|).$$

□

It is important to note that given a VCC and a CC of a graph $G$, *the edges crossing them in G are fixed. We hence index the crossing edges to optimize the construction of X* by augmenting the VP. Using the augmented VP, *we can use one scan on the crossing edges (Line 09 in Fig. 9) to determine all the pairs* $(I, I')$ *satisfying* $I \cap N(I') \neq \emptyset$ *for all I's of the VCC and all I'* s *of the CC*. The time complexity can be significantly reduced.

The idea of the augmented VP is to represent the VCC and each CC as a supernode and *label* the superedge between the VCC and each CC by the list of crossing edges in $G$ between the VCC and the CC. The augmented VP is defined as follows.

**Definition 3** Given a VP $\mathcal{P} = \{CC_1, CC_2, ..., CC_k, VCC\}$ of $G$, the augmented VP is a bigraph $\mathcal{VP}^+ = (V_{vcc} \cup V_{cc}, E, label)$.

- $V_{cc} = \{VCC\}$ and $V_{vcc} = \{CC_1, CC_2, ..., CC_k\}$;
- $(VCC, CC_i)$ in $E$ for each $CC_i$ in $V_{cc}$;
- Each edge $(VCC, CC_i)$ is associated with a label $label(VCC, CC_i) = \{(u, v) | u \in VCC, v \in CC_i, (u, v) \in G\}$.

**Example 5** Figure 7 shows the augmented VP $\mathcal{P}_1^+$ of the VP $\mathcal{P}_1$ in Fig. 3. The difference between $\mathcal{P}_1^+$ and $\mathcal{P}_1$ is that there is a labeled edge between the VCC and each CC. $label(VCC, CC_1)$, $label(VCC, CC_2)$ and $label(VCC, CC_3)$ are shown in Fig. 7.

Based on Definition 3, the construction algorithm of the augmented VP is shown in Fig. 8.

**Optimized bigraph construction algorithm using $\mathcal{P}^+$** The label $label(VCC, CC_i)$ in the augmented VP stores the edges of

**Procedure** AugVP_cons
**Input**: A graph $G$, component size $s$
**Output**: Augmented VP $\mathcal{P}^+$

... //Lines 01-11 are the same with Fig. 5
12 construct an empty bigraph $\mathcal{P}^+$
13 add $\{VCC\}$ of $\mathcal{P}$ as one part of vertices of $\mathcal{P}^+$
14 add the CCs $\{CC_1, CC_2, ..., CC_k\}$ of $\mathcal{P}$ as the other part of $\mathcal{P}^+$
15 add an edge $(VCC, CC_i)$ to $\mathcal{P}^+$ for each $CC_i$ in $\mathcal{P}$
16 set $label(VCC, CC_i) = \emptyset$ for each edge $(VCC, CC_i)$ of $\mathcal{P}^+$
17 **for each** edge $e$ of $G$
18 **if** $e$ has an end in $VCC$ and another end in some $CC_i$
19 add $e$ to $label(VCC, CC_i)$
20 **return** $\mathcal{P}^+$

**Fig. 8** Procedure AugVP_cons

**Procedure** CCB_cons // CCB means Color Combination Bigraph
**Input**: the color of $VCC$ $\mathcal{I}_{VCC}$, the color of $CC_i$ $\mathcal{I}_{CC_i}$, $\mathcal{P}^+$
**Output**: a bigraph $X_i$ for combing $\mathcal{I}_{VCC}$ and $\mathcal{I}_{CC_i}$

01 **for each** $I_a \in \mathcal{I}_{VCC}$
02 **for each** $v \in I_a$, $ID^{IS}_{VCC}(v) = a$
03 **for each** $I_b \in \mathcal{I}_{CC_i}$
04 **for each** $v \in I_b$, $ID^{IS}_{CC}(v) = b$
05 construct an empty bigraph $X_i$
06 **for each** $I_a \in \mathcal{I}_{VCC}$, insert a vertex $a$ to a part $V_1$ of $X_i$
07 **for each** $I_b \in \mathcal{I}_{CC_i}$, insert a vertex $b$ to the other part $V_2$ of $X_i$
08 **for each** $a$ of $V_1$ and $b$ of $V_2$, insert an edge $(a, b)$ to $X_i$
09 **for each** edge $(u, v)$ in $label(VCC, CC_i)$ in $\mathcal{P}^+$//this is the one scan
//suppose $u \in VCC$ and $v \in CC_i$
10 delete the edge $(ID^{IS}_{VCC}(u), ID^{IS}_{CC_i}(v))$ from $X_i$
11 **return** $X_i$

**Fig. 9** Procedure bigraph_cons

$G$ crossing $VCC$ and $CC_i$. Consider an edge $(u, v)$ of $G$, if an IS $I$ of $VCC$ contains $u$ and an IS $I'$ of $CC_i$ contains $v$, $I$ and $I'$ cannot be merged. Therefore, *our main idea is to use the edges in label*$(VCC, CC_i)$ *to filter the ISs that cannot be merged.*

In Fig. 9, we show the construction algorithm of the bigraph using the augmented VP. Given the augmented VP $\mathcal{P}^+$ and the colors of $VCC$ and $CC_i$, Lines 01-04 map a vertex $v$ of $G$ to the ID of the IS containing $v$. $ID^{IS}_{VCC}(v)$ is the ID of the IS of $VCC$ that contains $v$; and $ID^{IS}_{CC_i}(v)$ is the ID of the IS of $CC_i$ that contains $v$. Lines 06-10 compute $X_i$ of $\mathcal{I}_{VCC}$ and $\mathcal{I}_{CC_i}$. For each IS $I_a$ and $I_b$ in $\mathcal{I}_{VCC}$ and $\mathcal{I}_{CC_i}$, we add a vertex $a$ and a vertex $b$ to $X_i$, respectively (Lines 06-07). Line 08 inserts an edge $(a, b)$ to $X_i$ for each $I_a \in \mathcal{I}_{VCC}$ and $I_b \in \mathcal{I}_{CC_i}$. (Line 08 constructs a complete bigraph, but the edges in the complete bigraph are just candidates to be filtered.) For each edge $(u, v)$ stored in $label(VCC, CC_i)$, we delete the edge $(ID^{IS}_{VCC}(u), ID^{IS}_{CC_i}(v))$ from $X_i$ (Lines 09-10). Finally, Line 11 returns $X_i$.

We use the following example to show how the redundant scans of the crossing edges of the VCC and the CC can be eliminated using the augmented VP.

**Example 6** Consider the $VCC$ and $CC_2$ in Fig. 7. The ISs of $VCC$ and $CC_2$ are shown in Fig. 10(a). To construct the bigraph $X_2$, we first construct a complete bigraph $X_2$, as shown in Fig. 10(b). For $(v_{10}, v_{11})$ in $label(VCC, CC_2)$, we delete the edge $(I^{VCC}_1, I^{CC_2}_1)$ from $X_2$. For $(v_5, v_6)$ in $label(VCC, CC_2)$, we delete the edge $(I^{VCC}_0, I^{CC_2}_0)$ from $X_2$. The final bigraph $X_2$ is shown in Fig. 10(c). The crossing edges of $VCC$ and $CC_2$ are canned for one pass.

**Proposition 5** Given a VP $\mathcal{P} = \{CC_1, CC_2, ..., CC_k, VCC\}$ of $G$, the total time to construct the bigraphs for $VCC$ and each $CC$ is $O(|V(G)| + |E_{cross}| + \sum_{i=1}^{k} |\mathcal{I}_{CC_i}||\mathcal{I}_{VCC}|)$.

**Proof** Consider a CC $CC_i$ of $\mathcal{P}$, the time complexity is $O(|VCC| + |CC_i| + |label(VCC, CC_i)| + |\mathcal{I}_{VCC}||\mathcal{I}_{CC_i}|)$, where $O(|VCC|)$ is for Lines 01-02, $|CC_i|$) is for Lines 03-04, $O(|\mathcal{I}_{VCC}||\mathcal{I}_{CC_i}|)$ is for Lines 05-08 and $O(|label(VCC, CC_i)|)$ is for Line 10.

Note that Lines 01-02 do not need to repeat for each CC. Therefore, for all CCs of $\mathcal{P}$, the total time complexity is

$$O(|VCC| + \sum_{i=1}^{k} |CC_i| + |E_{cross}| + \sum_{i=1}^{k} |\mathcal{I}_{CC_i}||\mathcal{I}_{VCC}|)$$
$$= O(|V(G)| + |E_{cross}| + \sum_{i=1}^{k} |\mathcal{I}_{CC_i}||\mathcal{I}_{VCC}|)$$

□

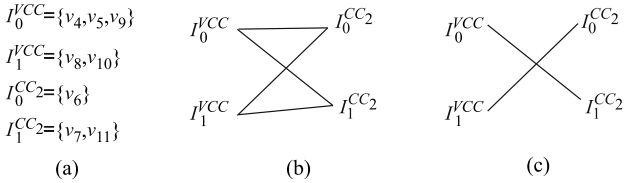


**Fig. 10** The process of generating $X_2$ for the color of $VCC$ $\mathcal{I}_{VCC}$ and the color of $CC_2$ $\mathcal{I}_{CC_2}$. (a) Colors of $VCC$ and $CC_2$; (b) complete bigraph $X_2$; (c) bigraph $X_2$

**Remarks** We can see that the time complexity is much smaller than that of the CCB construction in Lines 14-19 of comb of Fig. 4 (Proposition 4).

### 4.2 CC coloring optimization

This subsection optimizes the coloring of each connected component (CC), which is the bottleneck of the subroutine-(ii). In Procedure Color, for each connected component $CC$, we use the MIS enumeration based coloring method (the function colorCC_*by_MISE* shown in Fig. 4). It may take exponentially long running time of $|CC|$, despite a small theoretical approximation ratio. However, in this subsection, we illustrate that when $CC$ is small and sparse, we can readily adopt Greedy [14] that has time complexity $O(|CC|)$, and at the same time preserve the approximation ratio. Therefore, we propose a hybrid algorithm that optimizes colorCC_*by_MISE* without a loss of approximation performance.

We propose a hybrid algorithm on the basis of this observation as presented below.

**Proposition 6** Given a graph component $C$, when $\log|C| \geqslant \Delta_C$, the approximation ratio of using Greedy on $C$ is no worse than that of using the function colorCC_*by_MISE*.

**Proof** The approximation ratio of colorCC_*by_MISE* for coloring $C$ is $1 + \log|C|$. The approximation ratio of the greedy algorithm is $1 + \Delta_C$. Therefore, when $\log|C| \geqslant \Delta_C$, the approximation ratio of Greedy on $C$ is no worse than that of colorCC_*by_MISE*. □

Figure 11 shows the pseudo-code of the hybrid algorithm. The time complexity of Procedure ColorCC is the same as colorCC_*by_MISE*. In practice, the hybrid algorithm is almost always more efficient than the previous algorithm. In our experiments, the running time of graph coloring is reduced by ~10% on average.

**Example 7** Consider the VP $\mathcal{P}_2$ of $G_2$, shown in Fig. 3. $\mathcal{P}_2$ has four CCs: $CC_1$, $CC_2$, $CC_3$ and $CC_4$. Procedure ColorCC will color $CC_1$ using colorCC_*by_MISE* as $\log|CC_1| = \log 3 < \Delta_{CC_1} = 2$. Similarly, $CC_2$ and $CC_3$ are also colored

**Procedure** ColorCC
**Input**: A Connected Component $CC$
**Output**: A coloring of $CC$

01 **if** $\Delta_{CC} \leq \log|CC|$
02 **return** Greedy($CC$)
03 **else**
04 **return** colorCC_by_MISE($CC$)

**Fig. 11** Procedure ColorCC

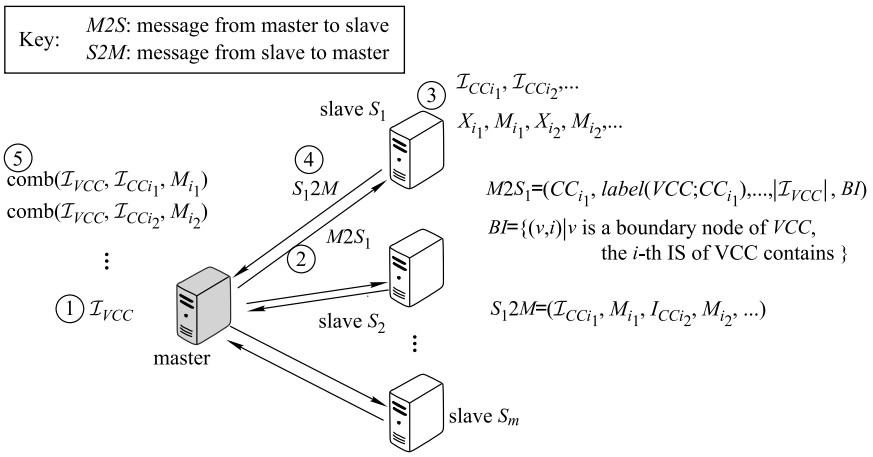


**Fig. 12** Illustration of the distributed graph coloring steps

by `colorCC_by_MISE`. Procedure `ColorCC` will color $CC_4$ by `Greedy` as $\log|CC_4| = \log 4 = \Delta_{CC_4} = 2$.

### 4.3 Distributed algorithm

Since each CC can be colored *independently* from other CCs and each MM can be computed *independently* from other MMs, we distribute them across multiple slave machines. The overall idea is that the coloring is CPU bounded in our framework. The network transfer just requires $L-1$ rounds, where $L$ is the height of `VPH`, and the messages are not very large. Therefore, there are benefits to distribute the coloring task.

Specifically, for the subroutine-(ii) (i.e., the component coloring subroutine), we color the CCs using different slaves. The VCC can be colored by the existing distributed graph coloring algorithms. For the subroutine-(iii) (i.e., the color combination subroutine), we split the subroutine to two tasks: (1) constructing CCB $X$ and computing the maximum matching (MM) of $X$, which is computed by the slaves; and (2) combining the colors of the VCC and the CC using the MM, which is computed by the master.

**Overview** The overview of the distributed graph coloring algorithm is illustrated with Fig. 12. It is implemented as a master-slave architecture. The circled numbers in Fig. 12 denote the operation steps. Given a VP $\mathcal{P}$ of $G$, ① the VCC is colored by the classical distributed coloring method and the master holds the coloring of the VCC. ② The CCs and the information of the coloring of the VCC are sent to the slaves. ③ The slaves color the CCs (as presented in Section 2), and compute the CCBs between the colors of the VCC and the CCs (as presented in Section 1) and compute the maximum matchings (MMs) of the CCBs. ④ The colorings of the CCs and the MMs are sent to the master. ⑤ The master combines the colorings of the VCC and the CCs using the MMs.

**Procedure on the master side** Figure 13 is the procedure for the master. The master has two tasks. The first task is to send messages to the slaves. The second task is to process the messages from the slaves. We will explain the procedure with a running example. In the example, we assume a cluster consisting of one master and two slaves $S_1$ and $S_2$ for simplicity of presentation.

The first task (Lines 03–06): the master first allocates the CCs to the slaves (Lines 18–22). Suppose $CC$ is the set of CCs

```
Procedure Master
Input: augmented VP P+ of a graph G, slave count slaveCount
Output: the coloring of G
01 let I_VCC be the coloring of VCC that is colored by the classical
       distributed algorithm
02 let I_CCi denote the coloring of CC_i
03 map = allocate_CCs(P+, slaveCount)
04 for each slave sid from 1 to slaveCount
05   M2S_sid = make_msg(VCC, I_VCC, map(sid))
06   master_send_msg(M2S_sid, sid) //send msg to the slave sid
07 while exist a slave not sent back message to master
08   S_sid2M = master_rev_msg(sid)
09   for each pair (I_CCi, M_CCi) in S_sid2M
10     for each I' ∈ I_CCi
11       if ∃(I, I') ∈ M_CCi, I ∈ I_VCC
12         I = I ∪ I'
13       else add I' to a list list_CCi
14   while exist non-empty list_CCi
15     pop the head I_CCi of each non-empty list_CCi
16     delete I_CCi from list_CCi
17     add the union of such I_CCi's to I_VCC
function allocate_CCs(P+, slaveCount)
18 initialize an empty map map, where the key of map is the
       slaveID and the value is the list of CCs allocated to the slave
19 for each CC_i in P+
20   slaveID sid = i%slaveCount
21   add CC_i to map(sid)
22 return map
function make_msg(VCC, I_VCC, map(sid))
23 initialize an empty msg
24 for each CC_j in map(sid)
25   add CC_j and label(VCC, CC_j) to msg
26 for each boundary node v of VCC
27   suppose v is in the i-th IS of I_VCC
28   add (v, i) to BI
29 add BI and |I_VCC| to msg
30 return msg
```

**Fig. 13** Procedure `Master`

allocated to the slave $sid$. For each $CC_i$ in $CC$, the master (Lines 04,05,23–25) adds $CC_i$, $label(VCC, CC_i)$ to $M2S_{sid}$. The master also adds the number of the ISs of $VCC$ and the boundary information of the ISs of $VCC$ (denoted by $BI$) to the slave $sid$. For $BI$, instead of sending all ISs of $VCC$, the master just sends *the boundary vertex $v$ and the ID of the IS containing $v$* to the slave. The boundary vertices are the vertices of $VCC$ that having edges outgoing $VCC$.

For example, consider the VP $\mathcal{P}_2$ of $G_2$ shown in Fig. 3. We assume $\mathcal{I}_{VCC} = \{\{v_4, v_8, v_{13}\}, \{v_9\}\}$ is the coloring of $VCC$, which is computed by the existing distributed coloring algorithm. For the first task, the master allocates $\{CC_1, CC_3\}$ and $\{CC_2, CC_4\}$ to $S_1$ and $S_2$, respectively. The master sends the message $M2S_1 = (CC_1, label(VCC, CC_1), CC_3, label(VCC, CC_3), 2, BI)$ to $S_1$, where 2 is the number of ISs in $\mathcal{I}_{VCC}$ and $BI = \{(v_4, 0), (v_8, 0), (v_9, 1), (v_{13}, 0)\}$, as $v_4, v_8, v_9$ and $v_{13}$ have edges outgoing $VCC$, and $v_4, v_8$ and $v_{13}$ are in the 0-th IS of $\mathcal{I}_{VCC}$ and $v_9$ is in the 1st IS of $\mathcal{I}_{VCC}$. The master sends to $S_2$ the message $M2S_2 = (CC_2, label(VCC, CC_2), CC_4, label(VCC, CC_4), 2, BI)$.

The second task (Lines 07–17): the master extracts the ISs of the CCs and the MMs in the messages $S2M$'s from the slaves, and merges the ISs of the VCC and the CCs using MMs. Specifically, after receiving the message $S_{sid}2M$ from the slave $sid$, for each pair $(\mathcal{I}_{CC_i}, M_{CC_i})$ in $S_{sid}2M$, the master (Lines 09–12) merges the ISs of $\mathcal{I}_{VCC}$ and $\mathcal{I}_{CC_i}$ marked by the MM $M_{CC_i}$. The ISs of the $CC_i$ that cannot be merged with the ISs of $VCC$ are added to a list $list_{CC_i}$ (Line 13). Then, Lines 14–17 retrieve the next IS $I_j$ from $list_{CC_j}$ of each $CC_j$ in $\mathcal{P}^+$ and add the union of the $I_j$'s to the merged result. Note that the union can be efficiently computed, because any two ISs $I_j$ in $list_{CC_j}$ and $I_{j'}$ in $list_{CC_{j'}}$ for $j \neq j'$ are non overlapping.

We continue the above example. Let $S_1 2M$ denote the message from the slave $S_1$, and let $I_i^{VCC}$ and $I_i^{CC}$ denote the $i$th IS of $VCC$ and $CC$, respectively. $S_1 2M = ((\mathcal{I}_{CC_1}, M_{CC_1}), (\mathcal{I}_{CC_3}, M_{CC_3}))$, where $\mathcal{I}_{CC_1} = \{\{v_1, v_3\}, \{v_2\}\}$, $\mathcal{I}_{CC_3} = \{\{v_{14}, v_{16}\}, \{v_{15}\}\}$, $M_{CC_1} = \{(I_1^{VCC}, I_0^{CC_1})\}$, $M_{CC_3} = \{(I_0^{VCC}, I_1^{CC_3}), (I_1^{VCC}, I_0^{CC_3})\}$. Using $M_{CC_1}$, the master combines $\mathcal{I}_{VCC}$ and $\mathcal{I}_{CC_1}$. $\mathcal{I}_{VCC}$ becomes $\{\{v_4, v_8, v_{13}\}, \{v_1, v_3, v_9\}\}$ and $list_{CC_1} = \{\{v_2\}\}$. Using $M_{CC_3}$, the master combines $\mathcal{I}_{VCC}$ and $\mathcal{I}_{CC_1}$. $\mathcal{I}_{VCC}$ becomes $\{\{v_4, v_8, v_{13}, v_{15}\}, \{v_1, v_3, v_9, v_{14}, v_{16}\}\}$ and $list_{CC_3} = \emptyset$. Finally, after processing $list_{CC_1}$ and $list_{CC_2}$, $\mathcal{I}_{VCC}$ becomes $\{\{v_4, v_8, v_{13}, v_{15}\}, \{v_1, v_3, v_9, v_{14}, v_{16}\}, \{v_2\}\}$. The processing of the message from $S_2$ is similar.

**Procedure on the slave side** Figure 14 is the procedure of the slave. The slave receives the message $M2S$ from the master. $M2S$ contains the information of the $VCC$ and a set of CCs. For each $CC_i$, the slave constructs the bigraph $X_i$ to compute the MM $M_{CC_i}$ of the colorings of $VCC$ and $CC_i$. The slave adds $|\mathcal{I}_{VCC}|$ vertices to one part of $X_i$, where each vertex denotes an IS of $VCC$ (Line 04); the slave adds a vertex to the other part of $X_i$ for each IS ID of $CC_i$ (Line 05); and the slave adds all possible edges to $X_i$ (Line 06). Then, $label(VCC, CC_i)$ in $M2S$ is used to compute the edges of $X_i$ (Lines 07–09). Finally, the maximum matching of $X_i$ is returned (Lines 10–12). We continue with the example above. $S_1$ receives $M2S_1$ from the master. $S_1$ colors $CC_1$ and $CC_3$ and obtains $\mathcal{I}_{CC_1}$ and $\mathcal{I}_{CC_3}$. For $CC_1$ and $CC_3$, $S_1$ constructs the bigraphs $X_{CC_1}$ and $X_{CC_3}$, and computes the maximum matching $M_{CC_1}$ and $M_{CC_3}$ of $X_{CC_1}$ and $X_{CC_3}$, respectively. The message $S_1 2M = \{(\mathcal{I}_{CC_1}, M_{CC_1}), (\mathcal{I}_{CC_3}, M_{CC_3})\}$ is returned to the master. The processing of $S_2$ is similar.

*Analysis of time complexity.* Using Procedures `Master` and `Slave`, the subroutines (ii) and (iii) are computed by the slaves in a distributed manner.

**Procedure** `Slave`
**Input**: the message $M2S$ from the master
**Output**: the message $S2M$

01 **for each** $CC_i$ in $M2S$
02 $\mathcal{I}_{CC_i}$ = color $CC_i$ by Procedure `ColorCC` in Figure 11
03 construct an empty bigraph $X = (V_1 \cup V_2, E)$
04 **for each** $a = 0$ to $|\mathcal{I}_{VCC}|$, insert a vertex $a$ to $V_1$ of $X$
05 **for each** IS $I_b$ in $\mathcal{I}_{CC_i}$, insert a vertex $b$ to $V_2$ of $X$
06 **for each** $a$ in $V_1$ and $b$ in $V_2$, insert an edge $(a, b)$ to $X$
07 **for each** edge $(u, u')$ in $label(VCC, CC_i)$ in $M2S$
08 **if** $\exists(u, a)$ in $BI$ of $M2S$ and $\exists I_b \in \mathcal{I}_{CC_i}$ contains $u'$
09 delete the edge $(a, b)$ from $X$
10 compute the MM $M_{CC_i}$ of $X$
11 add the pair $(\mathcal{I}_{CC_i}, M_{CC_i})$ to $S2M$
12 **return** $S2M$

**Fig. 14** Procedure `Slave`

**Proposition 7** Given the augmented VP $\mathcal{P}^+$ of a graph $G$, the time complexity of Procedure `Master` and Procedure `Slave` for coloring $G$ is $O(t_{vcc} + x(s^2 3^{s/3} + t_{CCB}^{opt} + \sqrt{2}\Delta_G^{2.5}) + |G|)$, where $t_{vcc}$ is the time complexity of the classical distributed algorithm for coloring the VCC, $x$ is the number of CCs processed by a slave and $t_{CCB}^{opt}$ is the time of the optimized CCB construction shown in Proposition 5.

**Proof** The time complexities of three parts are analyzed as follows. The time to color the VCC of $\mathcal{P}^+$ is simply $t_{vcc}$.

Suppose each slave processes $x$ CCs. It takes $O(xs^2 3^{s/3})$ to color the CCs, $O(xt_{CCB}^{opt})$ to construct the CCBs, and $O(x\sqrt{2}\Delta_G^{2.5})$ to compute the MMs of the CCBs.

The master computes the union of the ISs of the VCC and the ISs of the CCs. It takes $O(|G|)$ time.

Therefore, the total time complexity is $O(t_{vcc} + x(s^2 3^{s/3} + t_{CCB}^{opt} + \sqrt{2}\Delta_G^{2.5}) + |G|)$. □

Proposition 7 shows that the time complexity of our distributed graph coloring algorithm is smaller than that of the centralized algorithm (Proposition 3), as $x$ is smaller than the number of CCs.

## 5 Experimental evaluation

In this section, we present an experimental evaluation of `VColor*`.

**Experiment settings** The experiments of the centralized algorithms are conducted on a server with an Intel Xeon 2.67GHz CPU and 32GB RAM, running CentOS 5.6. For distributed algorithms, we use five servers with the above configuration in this experiment, where one server is used as the master and four servers are used as the slaves. We implement the algorithm in Java 1.7. The popular graph library `jgrapht` is used in our implementation.

**Benchmark datasets** We use four datasets in our experiments: two graphs of small size (`LS` and `Yeast`) and two graphs of large size (`PA` and `Pokec`). `LS` is a latin square graph, which is often used in graph coloring works [19]. `Yeast` is a biological network of Yeast [20]. `Pokec` is a social network. `PA` and `NY` are two road networks. They are available at Stanford Large Network Dataset Collection. As the original graph of `Pokec` is directed, we convert it into an undirected graph, by removing the directions of all edges. Table 3 reports some statistics of the graphs.

The running time is the total time of the VP construction and

**Table 3** Some statistics of benchmarked datasets

| | $\|V(G)\|$ | $\|E(G)\|$ | $\Delta_G$ |
|---|---|---|---|
| Pokec | 1.63M | 22.30M | 8,784 |
| PA | 1.09M | 1.54M | 9 |
| NY | 264K | 733K | 8 |
| Yeast | 3.1K | 12.5K | 168 |
| LatinSquare | 0.9K | 307.4K | 683 |

the coloring. For a graph $G$, if the VCC of the VP of $G$ is still large, we construct a VPH of $L$ levels by recursively partitioning the VCC. In this case, the running time is the total time of the VPH construction and the coloring.

### 5.1 Comparison with existing centralized algorithms

In this subsection, we compare VColor* with VColor [1], SampleIS [9], Greedy [14] and JP [21]. VColor is our recent work. SampleIS is the algorithm that currently has the best-known approximation ratio. Greedy is a greedy coloring algorithm. The idea of JP is summarized in Section 6. Since we focus on the centralized algorithms in this subsection, the techniques in Section 1 and Section 2 of VColor* are used in this experiment and the techniques in Section 3 are not. Similarly, the distribution technique of JP is not used as well.

#### 5.1.1 Comparison on graphs of small size

We first show the comparison results on the small graphs Yeast and LS, as SampleIS cannot finish on other graphs.

**Experiments on the coloring time** Figures 15(a)–(b) show the comparison of the running time on Yeast and LS, respectively. We can observe that Greedy is the fastest. However, VColor* is close to Greedy. Moreover, VColor* is about 30% and 50% faster than VColor on Yeast and LS, respectively.

**Experiments on the number of colors** Figures 15(c)–(d) show the comparison of number of colors on Yeast and LS, respectively. We can observe that VColor* uses the same number of colors with Greedy on Yeast, but uses fewer colors than Greedy on LS.

In sum, VColor* can use slightly longer time than but use fewer colors than Greedy on graphs of small size.

#### 5.1.2 Comparison on large graphs

In this subsection, we compare VColor*, VColor, Greedy and JP on large graphs Pokec, PA and NY. For VColor* and VColor, we show the smallest color number (obtained by tuning $s$ and $L$) and the corresponding coloring time. Table 4 shows the result. We can observe that Greedy is always the fastest. However, VColor* can use fewer colors by using more time. In particular, on Pokec, VColor* uses extra 483 seconds to save 13 colors in comparison with Greedy. On PA and NY, VColor* just saves 1 color in comparison with Greedy with the cost of using 18 and 28 more seconds, respectively. VColor* can be used in the scenario where the number of colors is very critical and tens of seconds is affordable. We also observe that VColor* uses the same number of colors with VColor, but VColor* is hundreds of times faster.

#### 5.1.3 Detailed comparison of VColor* and VColor

For VColor* and VColor, we construct VPHs as the graphs are large. We tune both $s$ and $L$ of the VPHs in this experiment.

**Experiments on the coloring time** In this experiment, we examine $L = 15$ and 20 on Pokec and $L = 3$ and 4 on PA, as the sizes of the VCC are small enough to be colored by SampleIS.

**Table 4** Comparison of coloring result on large graphs

| | Color number | | | Coloring time/s | | |
|---|---|---|---|---|---|---|
| | Pokec | PA | NY | Pokec | PA | NY |
| VColor* | 35 | 5 | 5 | 522 | 22 | 30 |
| VColor | 35 | 5 | 5 | 45300 | 4500 | 116 |
| Greedy | 48 | 6 | 6 | 39 | 3.7 | 1.4 |
| JP | 41 | 5 | 5 | 266.7 | 6.8 | 3.3 |

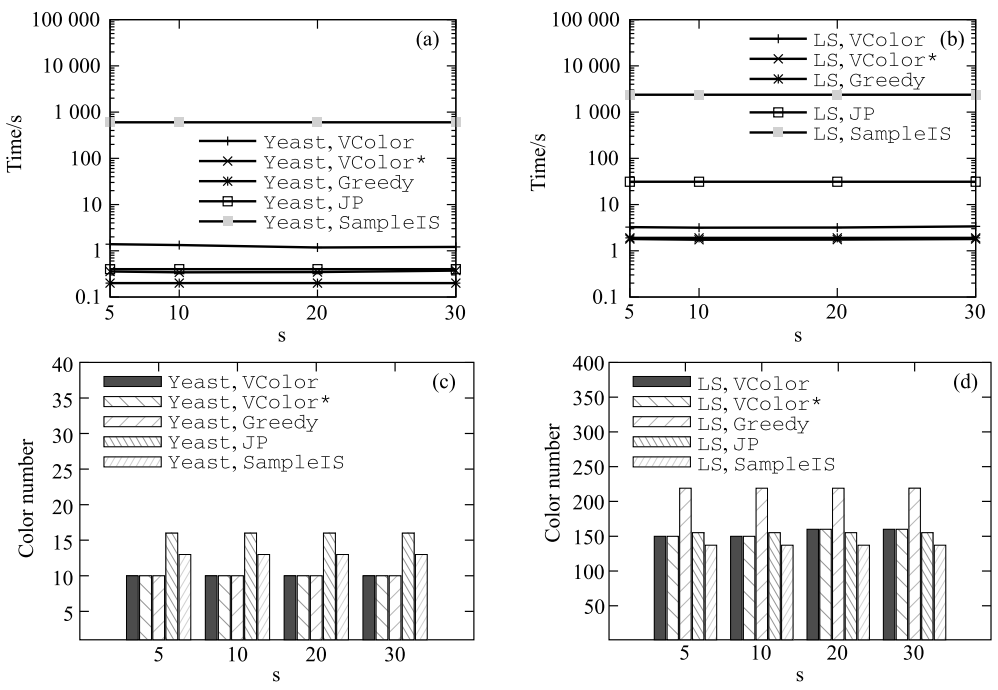


**Fig. 15** Results on LS and Yeast. (a) Coloring time on Yeast; (b) coloring time on LS; (c) color number on Yeast; (d) color number on LS

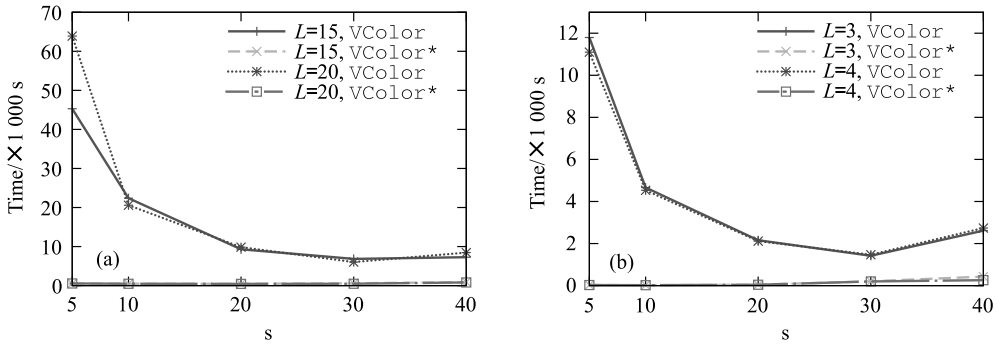


**Fig. 16** Time of coloring large graphs. (a) Coloring time on Pokec; (b) coloring time on PA

From Figs. 16(a)–(b), we observe that the coloring time first reduces and then increases with the growth of $s$. The main reason for this is that although the number of CCs decreases with the growth of $s$ but the time to color a CC increases. Figure 16(a) also shows that the coloring times for $L = 15$ and 20 are very close on Pokec. The reason is that the VPH for $L = 15$ is very close to that for $L = 20$. For example, when $s = 20$, the VPBs $B_{16}, B_{17}, ..., B_{20}$ contain only 1,851 vertices in total, which is very small compared with the size of Pokec. A similar observation is found for PA as shown in Figure 16(b). Figures 16(a)-(b) show that VColor* is significantly faster than VColor. In particular, on Pokec, when $L = 15$ and $s = 20$, VColor* is 20X faster than VColor. On PA, when $L = 3$ and $s = 20$, VColor* is 50X faster than VColor.

**Experiments on the number of colors** Following the above experiment, we also set $L = 15$ and 20 on Pokec and $L = 3$ and 4 on PA. The results are shown in Figure 17.

Figure 17(a) shows that the number of colors increases with the growth of $s$, but the marginal increase reduces on the social network. Figure 17(a) also shows that the number of colors slightly increases with the growth of $L$. The reason is that the VCC of $B_{15}$ is further partitioned for $L = 20$, and hence the VCC of $B_{15}$ can be colored using slightly more colors by VColor than by directly applying SampleIS on it. From Fig. 17(b), we observe that the number of colors is very stable with the growth of $s$ and $L$ on the road network PA. Figures 17(a)–(b) show that VColor* uses the same number of colors as VColor.

## 5.2 Performance of the optimizations on centralized VColor*

In this experiment, we focus on the centralized VColor* to show the effectiveness of the optimization techniques in Section 1 and Section 2. We use opt1 and opt2 to denote the optimization for constructing the CCB (Section 1) and that for coloring CCs (Section 2), respectively.

### 5.2.1 Effectiveness of opt1

**Speedup ratio** The speedup ratio of opt1, $SR_{opt1}$, is defined as $1 - t_{opt1}/t$, where $t_{opt1}$ and $t$ are the running time of VColor with and without opt1, respectively.

Figure 18(a) shows $SR_{opt1}$ on Pokec. From Fig. 18(a), we can observe that opt1 can significantly reduce the running time on Pokec. In particular, Figure 18(a) shows that $SR_{opt1}$ exceeds 95% and can be up to 99%.

Figure 18(a) also shows that $SR_{opt1}$ reduces as $s$ increases. The reason is that the number of crossing edges between the VCCs and the CCs reduces as $s$ increases. The main effect of opt1 is to reduce the time spent on the crossing edges. Hence, the speedup ratio reduces with $s$.

We can observe a gap between the speedup ratio of $L = 15$ and that of $L = 20$ from Fig. 18(a). The reason for this is that the VPH of $L = 20$ has more crossing edges between the VCCs and the CCs than the VPH of $L = 15$, as the VCCs of the 15th level of the VPH of $L = 15$ are further partitioned to the levels 16-20 of the VPH of $L = 20$.

We further observe from Fig. 18(a) that the gap between $L = 15$ and $L = 20$ increases with the growth of $s$. The reason for this is that the reduction of the number of the crossing edges of $L = 15$ is faster than that of $L = 20$.

**Coloring time** Figure 18(b) shows the running time of VColor+opt1 on Pokec. We can observe that the running time first decreases and then increases with the growth of $s$. It is consistent with the trend of VColor.

Figure 18(b) also shows that the best value of $s$ that produces the smallest running time becomes 20. Recall that the best value

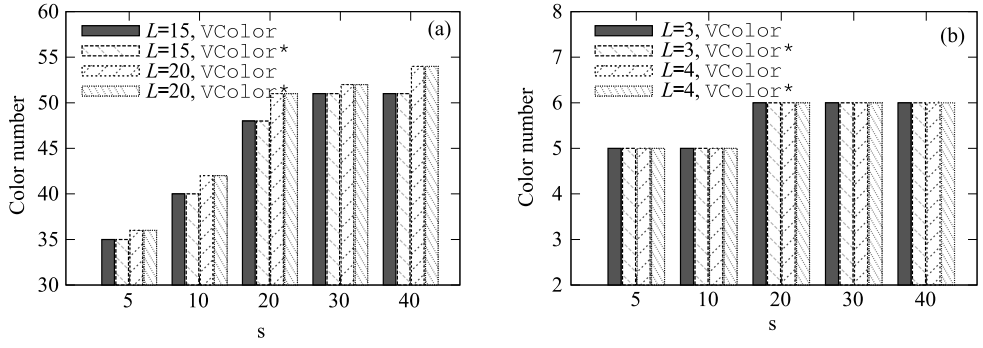


**Fig. 17** Color number on large graphs. (a) Color number on Pokec; (b) color number on PA

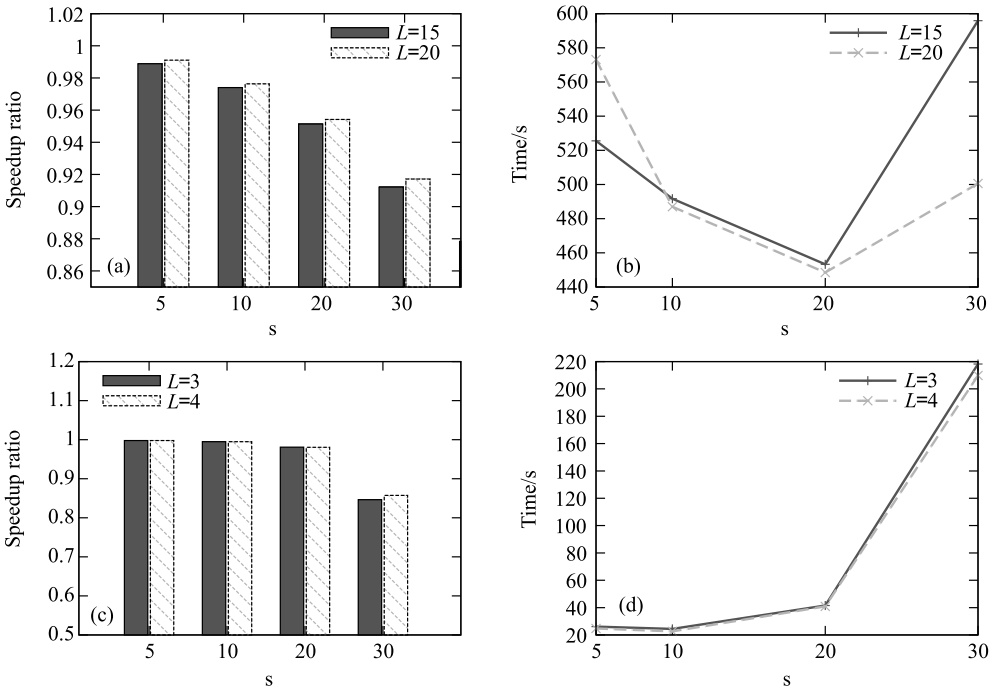


**Fig. 18** Time performance of opt1. (a) $SR_{opt1}$ on Pokec; (b) time of VColor+opt1 on Pokec; (c) $SR_{opt1}$ on PA; (d) time of VColor+opt1 on PA

of $s$ is 30 without opt1 as shown in Figure 16(a). The reason for this is that opt1 has effects on the crossing edges between the VCCs and the CCs, and the VPH of a smaller $s$ has more crossing edges. Therefore, the best value of $s$ can be smaller when using opt1.

Similar observations are found for PA as shown in Figs. 18(c)–(d).

#### 5.2.2 Effectiveness of opt2

**Speedup ratio** The speedup ratio of opt2, $SR_{opt2}$, is defined as $1 - t_{opt1,opt2}/t_{opt1}$, where $t_{opt1,opt2}$ is the running time of VColor+opt1+opt2.

Figure 19(a) shows $SR_{opt2}$ on Pokec. Figure 19(a) shows that $SR_{opt2}$ is relatively small when compared to $SR_{opt1}$. It is reasonable as opt2 is a very light-weight optimization. Figure 19(a) shows that $SR_{opt2}$ increases with the growth of $s$. The reason is that the difference between the running time of Greedy and the MIS enumeration based method is larger when CCs are larger.

We can also observe from Fig. 19(a) a gap between the speedup ratio of $L = 15$ and $L = 20$. It is because that the VPH of $L = 20$ has more CCs than the VPH of $L = 15$. Figure 19(a) shows that the gap increases with the growth of $s$ as Greedy is more efficient on larger CCs.

**Coloring time** Figure 19(b) shows the running time of VColor+opt1+opt2 on Pokec. We can observe that the trend of the running time is consistent with the running time of opt1 shown in Fig. 18(b). Different from opt1 that can change the best value of $s$, Figure 19(b) shows that opt2 does not change the best value of $s$, as the effect of opt2 is too small to change it.

Similar observations are found for PA as shown in Figs. 19(c)–(d).

#### 5.2.3 Color number

Figure 20 shows the color number after using the optimizations. We can observe that using the opt1 and opt2 will not increase the number of colors used.

### 5.3 Performance of distributed techniques in VColor*

In this experiment, VColor* involves the techniques in Section 1, Section 2 and Section 3. We compare VColor* with the widely used distributed graph coloring algorithm KW [15]. Since KW is based on the BSP model, we run KW on Apache Giraph, which is a popular system supporting the BSP model. We also compare with the distributed JP [21], where we use the technique in [22] to color the boundary nodes and then color the components independently by slaves.

**Results of VColor*** Figures 21(a)–(b) show the running time of our distributed graph coloring algorithm on Pokec and PA, respectively. From Fig. 21(a), we can observe that the running time reduces with the growth of the number of slaves. When one slave is used, the running time is longer than that of the centralized algorithm (Fig. 19). It is reasonable due to the network communication cost. However, when two or more slaves are used, the running time is smaller than that of the centralized algorithm. We can make similar observations on PA from Fig. 21(b). The number of colors used by our distributed graph coloring algorithm is the same as that used by the centralized algorithm.

**Results of KW** Figure 22 shows the running time and the number of colors used by KW on Pokec and PA. Figures 22(a)–(b) show that the running time reduces with the growth of the slave number, but increases with more iterations. We fix the slave number to be 4 and study the color number used. Figures 22(c)–(d) show that the color number reduces with the growth of the number of iterations.

From Fig. 22, we can observe that *our distributed method-*

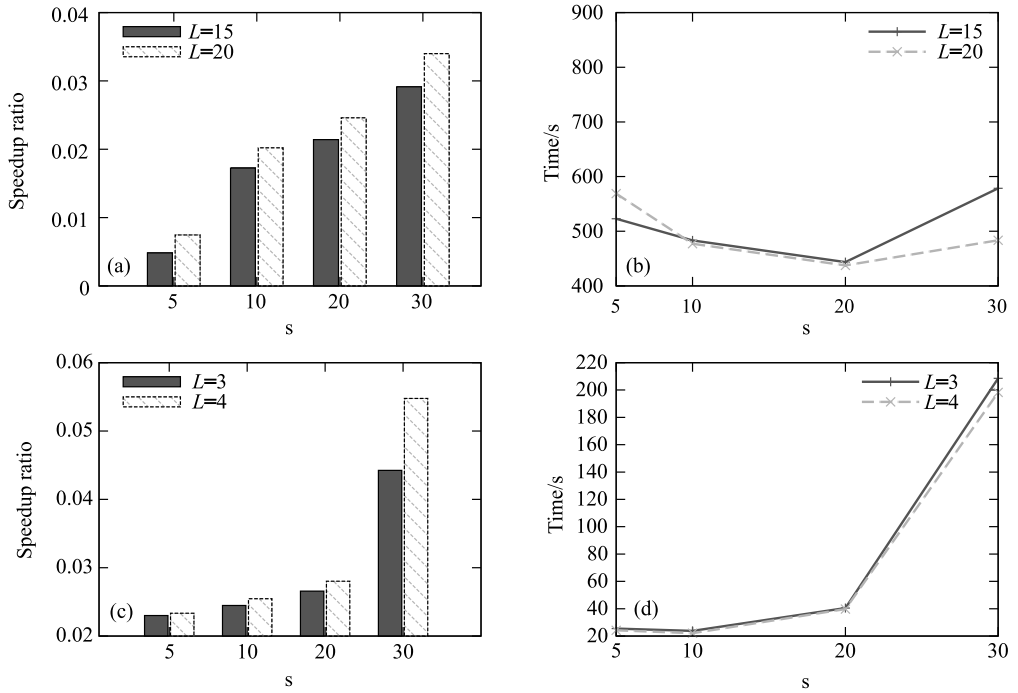


**Fig. 19** Time performance of opt2. (a) $SR_{opt2}$ on `Pokec`; (b) time of `VColor+opt1+opt2` on `Pokec`; (c) $SR_{opt2}$ on `PA`; (d) time of `VColor+opt1+opt2` on `PA`

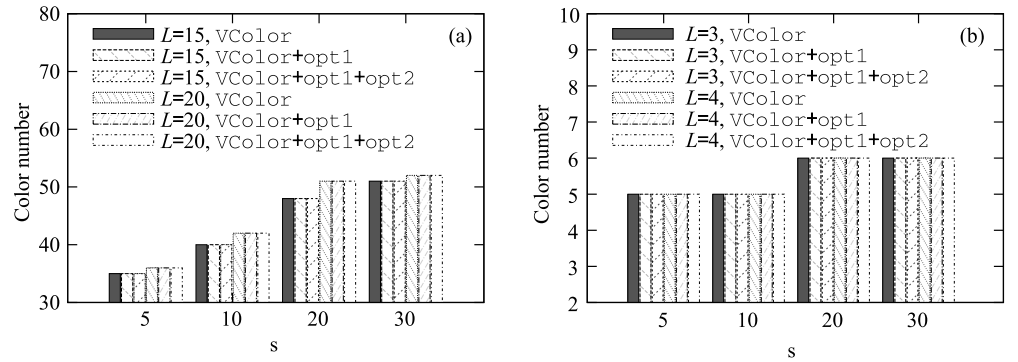


**Fig. 20** Color number performance of opt1 and opt2. (a) Color number on `Pokec`; (b) color number on `PA`

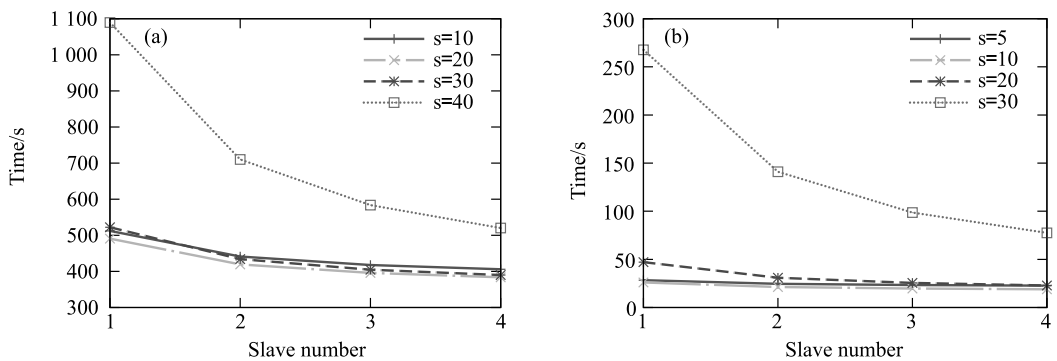


**Fig. 21** Time performance of distributed coloring. (a) Time on `Pokec`; (b) time on `PA`

*outperforms* `KW` *in both the color number and the running time*. For example, on `Pokec`, when we use 4 slaves and set the iteration number to be 100, `KW` uses 1,541,511 colors and takes about 844 seconds. In contrast, our distributed method just uses about *50* colors and takes about 400 seconds. Although we can reduce the color number of `KW` by using more iterations, the running time increases greatly and the color number reduces slightly with the growth of the number of iterations, as shown in Figs. 22(a) and (c). We can make similar observations on `PA` from Figs. 22(b) and (d).

**Results of `JP`** Figure 23 shows the running time and the number of colors used by `JP` on `Pokec` and `PA`. Figures 23(a)–(b) show that the running time reduces with more slaves. *However, the running time of* `JP` *is much longer than that of* `VColor*`

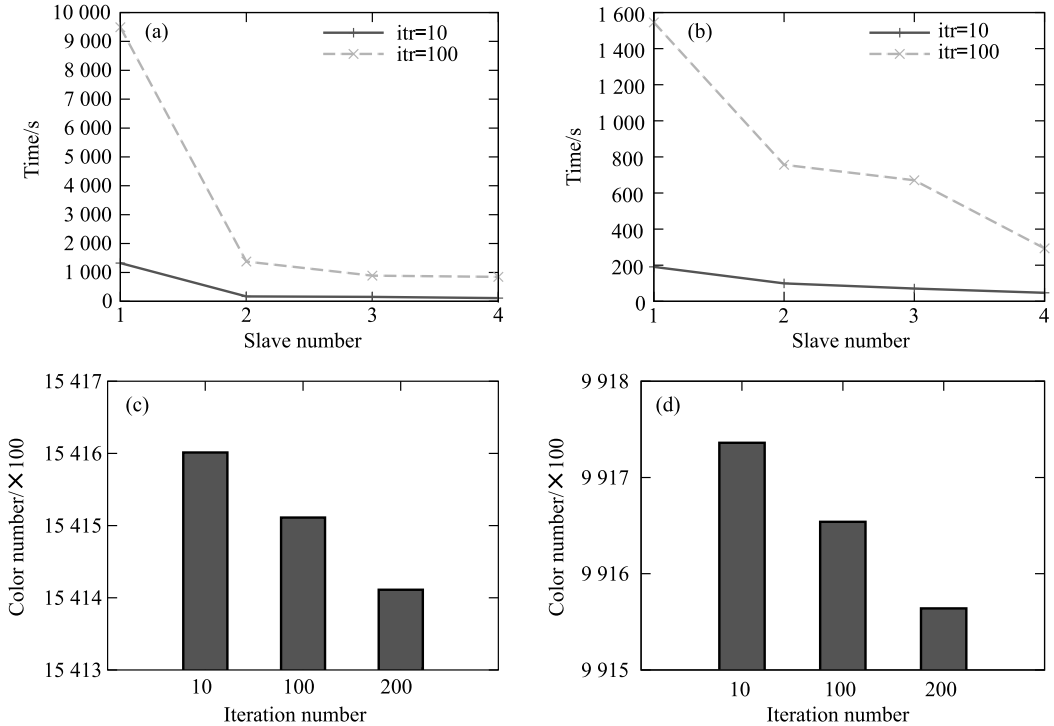


**Fig. 22** Performance of KW. (a) Time on Pokec; (b) time on PA; (c) color number on Pokec; (d) color number on PA

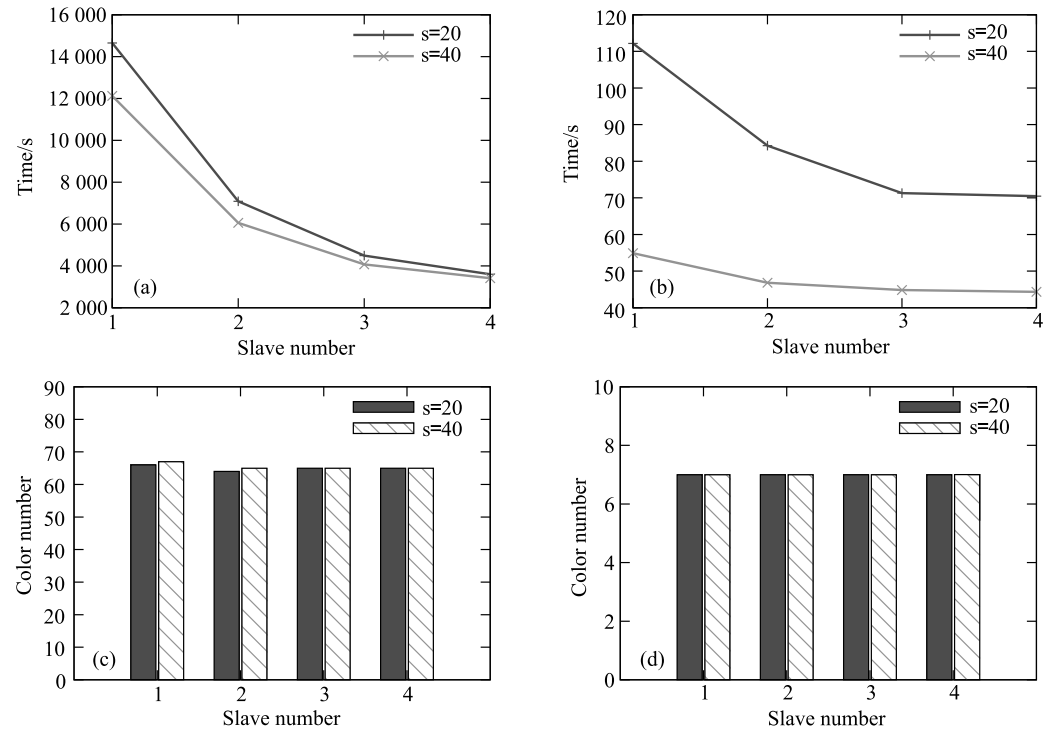


**Fig. 23** Performance of JP. (a) Time on Pokec; (b) time on PA; (c) color number on Pokec; (d) color number on PA

*(Fig. 21). One reason is that when coloring the boundary nodes, JP needs to repeatedly send the colors of the already colored boundary nodes over network for several times*. Moreover, the marginal reduction of the running time reduces with more slaves.

Figures 23(c)–(d) show the number of colors used on Pokec and PA, respectively. We can observe that JP uses more colors than VColor* (Fig. 20). One reason is that JP first colors the vertices in the boundary subgraph and then colors the vertices in the CCs, where the coloring of the CCs depends on the coloring of the boundary subgraph. However, such order may be far from the best order. In addition, the color of a vertex cannot change once the vertex is colored. In contrast, we color the boundary subgraph and the CCs independently. In addition, the

color combination step of VColor* can modify the colors of vertices.

## 6 Related work

In this section, we present the works that are closely related to this paper. We mainly discuss the inexact methods and the readers who are interested in exact solutions may refer to decent surveys [11, 13]. There are works study coloring the dynamic graphs (e.g., [23,24]). However, VColor* focuses on coloring the static graphs. It is possible that using VColor* to color a graph and using the dynamic algorithm to maintain the coloring after the graph is updated. However, it can be inefficient to use the dynamic algorithm to color a graph from scratch (i.e., starting from the coloring of an empty graph and maintain the coloring after the insertion of each edge), as shown in the experiments of [23].

We first discuss JP [21] as it is the most similar to our work. The similarity relies on that JP also partitions the graph into components, and colors the boundary subgraph and the components separately. Then, we discuss other related works.

The main idea of JP is as follows. Given a graph, JP first partitions the graph into components by an edge cut. Then, it colors the boundary nodes. Finally, for each component, the boundary nodes of the component has been colored and the remaining nodes of the component are colored by the first fit method (see Section 1). However, JP has two disadvantages as verified in our experiments. On one hand, JP uses more colors than our VColor*. On the other hand, while JP can be distributed, the distributed JP takes more time than VColor* as it repeatedly sends the color information of the already colored nodes several times over network.

### 6.1 Centralized graph coloring methods

Most existing approximation algorithms fall into three frameworks: the first fit approach, the local search approach, the evolutionary approach and the independent set extraction approach. They are discussed as follows.

**First fit** The main idea of the first fit method is to color a vertex using the smallest valid color, where the vertices are picked in a certain order. There are several well-known orders in the literature such as the arbitrary order, the largest degree first order, the smallest degree last order, the incidence degree order and the saturation degree order, etc. The first fit method guarantees to color a graph with at most $\Delta+1$ colors. However, [21] argues that it is hard to be computed in a distributed way.

**Local search** The main idea of local search is to iteratively change the color of a vertex that can decrease the value of a cost function, until a local optimum is reached. A tabu algorithm TABUCOL [8] is a seminal work of local search and there are many innovative variations. For example, Blöchliger and Zufferey [25] design a dynamic tabu algorithm to better capture the neighborhood changes throughout the search. Porumbel et al. [26] modify the cost function of TABUCOL by assigning different costs to different edge violations. Hertz et al. [27] design a hybrid method, which integrates TABUCOL and its two variations. The hybrid method outperforms the individual variations in many circumstances. There are also some simulated annealing algorithms [28]. While the local search algorithms generally support coloring on small and modest size graphs well, their results are often far from the optimum on large graphs [11].

**Evolutionary approach** The main idea is to use colorings as individuals of a generation of candidate solutions and to cross the individuals to pass good information to the offsprings. The crossing operator and the fitness function are crucial. Earlier works use the standard uniform crossover for crossing. For example, Fleurent and Ferland [29] assign to a vertex the color of either the first parent or the second parent. However, the evolutionary approach has not been as competitive as local search, until the GPX method [10] is proposed. Instead of using the color of a vertex as an individual, GPX proposes to use a color class as an individual and passes the color classes to offsprings in an alternating manner. This idea is effective and many recent works follow it. For example, Galinier et al. [30] propose to combine conflict-free color classes from parents. When selecting color classes to pass to offsprings, Porumbel et al. [31] propose to consider both the sizes of color classes and the conflicts. Lu and Hao [32] propose to use several parents in evolution. Evolutionary methods can produce good coloring results on large graphs. But, they are often time consuming.

**Independent set (IS) extraction** It is the most widely adopted framework for coloring large graphs [11]. SampleIS [9] used in our experiments follows this framework. This framework comprises two phases: a preprocessing phase and a coloring phase. The preprocessing phase is to iteratively extract a large IS from the input graph until the residual graph is small enough. Each IS extracted is assigned a unique color. The coloring phase uses existing methods (e.g., TABUCOL) to color the residual graph. The color classes of the residual graph and the ISs extracted give a coloring of the input graph. Many methods for computing large ISs have been proposed, such as simple greedy [33], tabu search [29], XRLF heuristic [34] and sampling [9]. To obtain a smaller residual graph, Wu and Hao [35] propose to extract a set of disjoint independent sets in each iteration, instead of extracting one independent set in each iteration. Recently, there is a trend of introducing a post processing phase, which reconsiders the color of each vertex [36, 37]. The main idea is to add back the ISs extracted to the residual graph and re-color the residual graph starting with its current coloring extended with the added ISs as new color classes.

Some works do not belong to the three frameworks. Karger et al. [12] model the graph coloring problem by semidefinite programming. It can color an $\alpha$-colorable graph with $\min\{\tilde{O}(\Delta^{1-2/\alpha}), \tilde{O}(|G|^{1-3/(\alpha+1)})\}$ colors. Although the bound is better than SampleIS, it is not definite. Karger et al. mainly focus on $\alpha$-colorable graphs and acknowledge that SampleIS has the best approximation ratio for general graphs.

### 6.2 Distributed graph coloring methods

Distributed graph coloring algorithms have also attracted a lot of attention in recent decades. There are randomized distributed algorithms (*e.g.*, [38]) and deterministic algorithms in the literature. In this paper, we focus on the deterministic algorithms.

JP can be distributed by first coloring the boundary nodes distributedly and then color each component independently [22, 39]. However, the distributed JP has a high network communication cost. Cole and Vishkin [40] propose a 3-coloring algorithm with time complexity $O(\log^* n)$ for oriented cycles,

where $n$ is the size of the input graph. Linial et al. [41] propose a $\log^* n + O(1)$ time $O(\Delta)$-coloring algorithm on general graphs. Linial et al. propose a lower-bound $\frac{1}{2}\log^* n - O(1)$ for the time complexity of the $f(\Delta)$-coloring algorithms, for any function $f(\Delta)$. Szegedy and Uishwanathan [42] improve the lower-bound to $\frac{1}{2}\log^* n + O(1)$ and propose a $O(\Delta^2)$-coloring algorithm. For the locally iterative algorithms, Szegedy et al. argue that no $(\Delta + 1)$-coloring algorithm can terminate in less than $\Omega(\Delta \log \Delta)$ time. Most of the currently known deterministic distributed graph coloring algorithms are locally iterative algorithms. KW [15] is the best known iterative algorithm. KW can color a graph with $\Delta + 1$ colors in $O(\Delta \log \Delta + \log^* n)$ time. Other coloring method includes the follows. Panconesi and Srinivasan [43] propose a $(\Delta + 1)$-coloring algorithm with time complexity $2^{O(\sqrt{\log n})}$. However, the network message cost is high [44]. Barenboim et al. [44] propose a $(\Delta + 1)$-coloring algorithm with time complexity $O(\Delta) + \frac{1}{2}\log^* n$. However, it only has an advantage on the graphs of $\Delta = o(\log n)$. Checco and Leith [45] study the distributed graph coloring algorithm to address the allocation task in the wireless network. However, the graph coloring algorithm studied in [45] is imposed strong constraints due to the physical limitations of the wireless equipment. For example, the access points in the wireless network cannot communicate with each other reliably, and hence vertices in the graph cannot share messages in the coloring algorithm. VColor* studies the general graph coloring problem and has no such constraint. [45] and VColor* have different focus.

## 7 Conclusion

In this paper, we propose VColor* that optimizes the divide-and-conquer framework of graph coloring [1]. The framework partitions a graph $G$ into a set of CCs and a VCC. The CCs and the VCC are colored separately. VColor* combines the local colors by an optimized maximum matching based method. VColor* proposes to color the sparse CCs by a greedy algorithm, which preserves the approximation ratio. VColor* also proposes a distributed graph coloring algorithm. Our experiments verify that VColor* is more efficient than the method in [1], while it does not use more colors.

**Acknowledgements** Thanks to the support of NSF of China (61773167, 61929103), NSF of Shandong Province (ZR2019LZH006), NSF of Shanghai (17ZR1444900, HKRGC GRF 12201119, 12232716 and 12201518), QLUT Young Scholar Program (2018DBSHZ005).

## References

1. Peng Y, Choi B, He B, Zhou S, Xu R, Yu X. Vcolor: a practical vertex-cut based approach for coloring large graphs. In: Proceedings of IEEE International Conference on Data Engineering. 2016, 97–108
2. Ingrid A. Nucleic acid sequence design as a graph colouring problem. Thesis, Universitat Wien, 2005, 1–83
3. Park T, Lee C Y. Application of the graph coloring algorithm to the frequency assignment problem. Journal of the Operations Research of Japan, 1996, 39(2): 258–265
4. Chaitin G. Register allocation and spilling via graph coloring. ACM Sigplan Notices, 1982, 17(6): 98–101
5. Lotfi V, Sarin S. A graph coloring algorithm for large scale scheduling problems. Computers & Operations Research, 1986, 13(1): 27–32
6. Moradi F, Olovsson T, Tsigas P. A local seed selection algorithm for overlapping community detection. In: Proceedings of IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining. 2014, 1–8
7. Yuan L, Qin L, Lin X, Chang L, Zhang W. Diversified top-k clique search. In: Proceedings of IEEE International Conference on Data Engineering. 2015, 387–398
8. Hertz A, De Werra D. Using tabu search techniques for graph coloring. Computing, 1987, 39(4): 345–351
9. HalldÃşrsson M M. A still better performance guarantee for approximate graph coloring. Information Processing Letters, 1993, 45(1): 19–23
10. Galinier P, Hao J K. Hybrid evolutionary algorithms for graph coloring. Journal of Combinatorial Optimization, 1999, 3(4): 379–397
11. Galinier P, Hamiez J P, Hao J K, Porumbel D. Recent advances in graph vertex coloring. Handbook of Optimization, 2013, 38: 505–528
12. Karger D, Motwani R, Sudan M. Approximate graph coloring by semidefinite programming. Journal of the ACM, 1998, 45(2): 246–265
13. Pardalos P M, Mavridou T, Xue J. The Graph Coloring Problem: a Bibliographic Survey. Handbook of Combinatorial Optimization, Springer, Boston, 1999, 1077–1141
14. Husfeldt T. Graph Colouring Algorithms. Cambridge University Press, 2015, 277–303
15. Kuhn F, Wattenhofer R. On the complexity of distributed graph coloring. In: Proceedings of ACM Symposium on Principles of Distributed Computing. 2006, 7–15
16. Eppstein D. All maximal independent sets and dynamic dominance for sparse graphs. ACM Transactions on Algorithms, 2009, 5(4): 1–14
17. Byskov J M. Enumerating maximal independent sets with applications to graph colouring. Operations Research Letters, 2004, 32(6): 547–556
18. Feige U, Mahdian M. Finding small balanced separators. In: Proceedings of the Annual ACM Symposium on Theory of Computing. 2006, 375–384
19. Johnson D J, Trick M A. Cliques, Coloring, and Satisfiability. Boston, MA, USA: American Mathematical Society, 1996
20. Lee J, Han W S, Kasperovics R, Lee J H. An in-depth comparison of subgraph isomorphism algorithms in graph databases. Proceedings of the VLDB Endowment, 2012, 6(2): 133–144
21. Jones M T, Plassmann P E. A parallel graph coloring heuristic. SIAM Journal on Scientific Computing, 1993, 14(3): 654–669
22. Salihoglu S, Widom J. Optimizing graph algorithms on pregel-like systems. Proceedings of the VLDB Endowment, 2014, 7(7): 577–588
23. Yuan L, Qin L, Lin X, Chang L, Zhang W. Effective and efficient dynamic graph coloring. Proceedings of the VLDB Endowment, 2017, 11(2): 338–351
24. Barba L, Cardinal J, Korman M, Langerman S, Renssen V A, Roeloffzen M, Verdonschot S. Dynamic graph coloring. Algorithmica, 2019, 81(4): 1319–1341
25. Blöchliger I, Zufferey N. A graph coloring heuristic using partial solutions and a reactive tabu scheme. Computers & Operations Research, 2008, 35(3): 960–975
26. Porumbel D, Hao J K, Kuntz P. A study of evaluation functions for the graph k-coloring problem. Artificial Evolution, 2008, 4926: 124–135
27. Hertz A, Plumettaz M, Zufferey N. Variable space search for graph coloring. Discrete Applied Mathematics, 2008, 156(13): 2551–2560
28. Morgenstern C, Shapiro H. Coloration neighborhood structures for general graph coloring. In: Proceedings of Annual ACM-SIAM Symposium on Discrete Algorithms. 1990, 226–235
29. Fleurent C, Ferland J. Genetic and hybrid algorithms for graph coloring. Annals of Operations Research, 1996, 63(3): 437–461
30. Galinier P, Hertz A, Zufferey N. An adaptive memory algorithm for the k-coloring problem. Discrete Applied Mathematics, 2008, 156(2): 267–279
31. Porumbel D C, Hao J K, Kuntz P. An evolutionary approach with diversity guarantee and well-informed grouping recombination for graph coloring. Computers & Operations Research, 2010, 37(10): 1822–1832
32. Lü Z, Hao J K. A memetic algorithm for graph coloring. European Journal

of Operational Research, 2010, 203(1): 241–250
33. Chams M, Hertz A, De Werra D. Some experiments with simulated annealing for coloring graphs. European Journal of Operational Research, 1987, 32(2): 260–266
34. Johnson D S, Aragon C R, McGeoch L A, Schevon C. Optimization by simulated annealing: an experimental evaluation; part ii, graph coloring and number partitioning. Operations Research, 1991, 39(3): 378–406
35. Wu Q, Hao J K. Coloring large graphs based on independent set extraction. Computers & Operations Research, 2012, 39(2): 283–290
36. Wu Q, Hao J K. An extraction and expansion approach for graph coloring. Asia-Pacific Journal of Operational Research, 2013, 30(5): 1350018
37. Hao J K, Wu Q. Improving the extraction and expansion method for large graph coloring. Discrete Applied Mathematics, 2012, 160(16–17): 2397–2407
38. Schneider J, Wattenhofer R. A new technique for distributed symmetry breaking. In: Proceedings of ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing. 2010, 257–266
39. Luby M. A simple parallel algorithm for the maximal independent set problem. In: Proceedings of Annual ACM Symposium on Theory of Computing. 1985, 1–10
40. Cole R, Vishkin U. Deterministic coin tossing with applications to optimal parallel list ranking. Information and Control, 1986, 70(1): 32–53
41. Linial N. Locality in distributed graph algorithms. SIAM Journal on Computing, 1992, 21(1): 193–201
42. Szegedy M, Vishwanathan S. Locality based graph coloring. In: Proceedings of ACM Symposium on Theory of Computing. 1993, 201–207
43. Panconesi A, Srinivasan A. On the Complexity of Distributed Network Decomposition. Academic Press, Inc., 1996
44. Barenboim L, Elkin M, Kuhn F. Distributed (delta + 1)-coloring in linear (in delta) time. Siam Journal on Computing, 2014, 43(1): 72–95
45. Checco A, Leith D J. Fast, responsive decentralized graph coloring. IEEE/ACM Transactions on Networking, 2017, 25(6): 3628–3640

Yun Peng received the PhD degree from the Hong Kong Baptist University, China in 2013 and received the BSci and MPhil degrees in computer science from Shandong University and the Harbin Institute of Technology (HIT), China in 2006 and 2008, respectively. His research interests include graph-structured data processing, data mining and machine learning.

Xin Lin received the BEng and PhD degrees, both in computer science and engineering, from Zhejiang University, China. He is currently a professor in the Department of Computer Science and Technology, East China Normal University, China. His research interests include location-based services, spatial databases, privacy-aware computing, and crowdsourcing.

Byron Choi is an associate professor in the Department of Computer Science at the Hong Kong Baptist University, China. He received the bachelor of engineering degree in computer engineering from the Hong Kong University of Science and Technology (HKUST), China in 1999 and the MSE and PhD degrees in computer and information science from the University of Pennsylvania, USA in 2002 and 2006, respectively.

Bingsheng He received the bachelor degree in computer science from Shanghai Jiao Tong University (1999–2003), and the PhD degree in computer science in Hong Kong University of Science and Technology (2003–2008), China. He is an associate professor in School of Computing, National University of Singapore, Singapore. His research interests are high performance computing, distributed and parallel systems, and database systems.