universität wien

MASTERARBEIT | MASTER'S THESIS

Titel | Title

Color Update Propagation: Dynamic Color Refinement in Evolving Graphs

verfasst von | submitted by

lic. Julia Korol

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien | Vienna, 2026

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 066 821
Studienrichtung lt. Studienblatt | Degree programme as it appears on the student record sheet:	Masterstudium Mathematik
Betreut von | Supervisor:	Assoz. Prof. Dipl.-Inf. Dr. Nils Morten Kriege

Abstract

Color refinement is a classical graph algorithm, also known as naive vertex classification or, more formally, the 1-dimensional Weisfeiler–Leman algorithm. It connects graph theory with group theory by partitioning a graph's vertices into color classes. Even a small structural modification, such as the addition or removal of a vertex or edge, can produce a very different refinement. In practice, however, such changes often influence only a limited subset of vertices, while the global partition is largely preserved. This raises the question: can the original vertex colors be used to efficiently obtain the updated partition after a graph modification?

In this thesis, I propose a dynamic variant of color refinement, called *Color Update Propagation*. Instead of restarting the refinement from scratch after each modification, this approach exploits the history of the original process and updates only those parts of the graph actually affected by the change.

I introduce three strategies for identifying the affected region. The most naive method updates the entire neighborhood of the modification, while more refined approaches exploit preserved symmetries to exclude unaffected vertices, thereby reducing computational effort.

Experimental results show that this approach performs well across a variety of graphs and their changes. As part of the thesis, I provide an implementation of the proposed algorithm, based on a variant of color refinement known as Power Iterated Color Refinement.

Zusammenfassung

Color Refinement ist ein klassischer Algorithmus auf Graphen, auch bekannt als naive Vertexklassifikation oder, formaler, als der eindimensionale Weisfeiler-Leman-Algorithmus. Er verbindet Konzepte aus der Graphentheorie und Gruppentheorie, indem er die Knoten eines Graphen in Farbklassen partitioniert. Bereits kleine strukturelle Änderungen, wie das Hinzufügen oder Entfernen einer Kante oder eines Knotens, können zu einer deutlich anderen Verfeinerung führen. In der Praxis betreffen solche Änderungen jedoch häufig nur einen begrenzten Teil der Knoten, während die globale Partition weitgehend erhalten bleibt. Dies führt zu der Frage, ob sich die ursprüngliche Färbung nutzen lässt, um die aktualisierte Partition nach einer Änderung effizient zu bestimmen.

In dieser Arbeit wird eine dynamische Variante von Color Refinement vorgestellt, die als *Color Update Propagation* bezeichnet wird. Anstatt den Verfeinerungsprozess nach jeder Änderung vollständig neu zu starten, nutzt dieser Ansatz die Informationen aus dem ursprünglichen Verlauf und aktualisiert nur diejenigen Teile des Graphen, die tatsächlich von der Änderung betroffen sind.

Es werden drei Strategien zur Bestimmung des betroffenen Bereichs untersucht. Die einfachste Methode aktualisiert die gesamte Nachbarschaft der Änderung, während weiterentwickelte Ansätze erhaltene Symmetrien ausnutzen, um unveränderte Knoten auszuschließen und so den Rechenaufwand zu reduzieren.

Experimentelle Ergebnisse zeigen, dass dieser Ansatz für verschiedene Graphklassen und unterschiedliche Arten von Änderungen gute Leistungen erzielt. Im Rahmen der Arbeit wird zudem eine Implementierung des vorgeschlagenen Algorithmus bereitgestellt, die auf einer Variante von Color Refinement basiert, dem sogenannten Power Iterated Color Refinement.

Contents

1 Introduction **4**

2 Notation **6**

3 Dynamic graph algorithms **7**

4 Color refinement and its properties **9**
4.1 Definitions and properties . . . 9
4.2 Complexity . . . 14

5 Algorithms **15**
5.1 Problem definition . . . 15
5.2 Color Update Propagation (CUP) . . . 16
5.2.1 Pseudocode . . . 17
5.2.2 Correctness . . . 18
5.2.3 Complexity . . . 19
5.3 CAS: pruning unchanged asymmetric vertices . . . 23
5.3.1 Pseudocode . . . 25
5.3.2 Correctness . . . 25
5.3.3 Complexity . . . 26
5.4 COR: pruning unchanged orbits . . . 27
5.4.1 Pseudocode . . . 28
5.4.2 Correctness . . . 29
5.4.3 Complexity . . . 29
5.5 Comparison of CUP, CAS, and COR . . . 31

6 Application **33**
6.1 Experiment Setting . . . 33
6.2 Random Trees: removing an edge . . . 34
6.2.1 Total proceeded vertices . . . 34
6.2.2 Average queue length . . . 35
6.2.3 Changed vertices in the final coloring . . . 36
6.2.4 Number of iterations . . . 39
6.3 G(n,p) Random Graphs . . . 39
6.3.1 Total processed vertices . . . 40
6.3.2 Number of iterations . . . 41
6.3.3 Average queue lengths through the refinement process . . . 42
6.3.4 Changed vertices in the final coloring . . . 43
6.4 Summary . . . 45

7 Conclusion and future work **46**

Appendix **48**

1 Introduction

Many graph problems require identifying vertices that play the same structural role. In such cases, vertices with the same position in the network should be treated identically, and graphs with the same structure should be recognized as equivalent. A common tool for capturing this idea is *color refinement* (CR), also known as the one-dimensional Weisfeiler-Leman (1-WL) procedure.

The algorithm starts with an initial coloring of the vertices and then repeatedly refines it using information from the neighborhoods. In each round, two vertices receive the same color if they had the same color in the previous round and the colors of their neighbors form the same multiset. The process continues until it stabilizes, that is, until two consecutive rounds induce the same partition. The resulting partition is called the *coarsest equitable partition*. It captures structural information about the graph and can be computed efficiently using existing tools, e.g. [1].

It was originally introduced by Weisfeiler and Leman in 1968 as a practical, but incomplete test for the graph isomorphism problem [2], and it remains a standard subroutine in many isomorphism testing algorithms.

CR is very effective and distinguishes almost all graphs; more precisely, the fraction of graphs with n vertices distinguished by CR converges to 1 as n grows to infinity [3]. However, it also has well-known limitations. In particular, it fails on graphs with a high degree of regularity. A simple example is given by a single cycle and a disjoint union of two cycles of the same length: in both graphs every vertex has exactly the same neighborhood structure, and CR assigns the same color to all vertices. Despite these limitations, CR is very efficient and is used as a preprocessing step in graph isomorphism solvers such as NAUTY [4].

Besides the graph isomorphism problem, color refinement is actively studied and applied in other areas. For example, it is used for reducing the dimensionality of systems of linear equations and linear programs by identifying symmetric variables and constraints [5]. In machine learning, color refinement is used in graph kernels [6], and its connections to graph neural networks have also been studied [7,8]. Finally, CR and its extensions are studied in finite model theory and logic, where they are generalized from graphs to relational structures [9].

The running time of color refinement has improved significantly since it was first introduced. The simplest implementation follows the refinement rounds directly. Since there can be at most n rounds and each round requires scanning all m edges, this leads to a running time of $O(nm)$. Later, Hopcroft introduced the technique of processing the smaller half [10]. This idea was adapted to graph partition refinement by Paige and Tarjan [11], resulting in an algorithm running in $O(m \log n)$ time, which is usually considered the standard running time for color refinement on static graphs [12].

In many applications, graphs are not static. Networks representing social media interactions, financial transactions, or relations between organisms in ecosystems change over time as new connections appear and old ones disappear.

Algorithms designed to operate in such changing environments are called *dynamic algorithms*. A dynamic algorithm maintains certain properties of a graph while the graph itself changes, for example through edge insertions or deletions. Compared to a static algorithm, the goal is to update the existing solution after each modification instead of recomputing the result from scratch, typically by reusing information from the previous state.

Dynamic graph algorithms have been studied for a wide range of problems, for example testing connectivity [13], finding shortest paths [14, 15], or calculating betweenness centrality [16]. For an overview of recent progress in this area we refer to the survey by Hanauer, Henzinger, and Schulz [17].

In this thesis we study color refinement from the dynamic perspective. This places the problem in the setting of dynamic graph algorithms, where the goal is to maintain a solution under updates. We consider updates in which a single edge is inserted into or deleted from the graph. After such a modification the existing coloring may no longer be correct. A straightforward solution is to recompute the refinement from scratch. However, since a small change in the graph often affects only a limited part of the structure, this raises the question whether the coloring can be updated more efficiently by reusing information from the previous refinement.

To the best of our knowledge, there is no commonly used dynamic algorithm for maintaining color refinement under such updates. In this thesis we therefore study this problem and propose three dynamic algorithms for updating color refinement after local changes to the graph.

We introduce a basic update procedure, *Color Update Propagation* (CUP), which propagates the effect of a change through the graph. We then propose two refinements of this approach, denoted CAS and COR, which use structural properties of the refinement to reduce unnecessary computations. For each algorithm we analyze correctness and complexity, and evaluate their practical behavior experimentally.

The thesis is organized as follows. Chapter 2 introduces the notation and basic definitions used throughout the thesis. Chapter 3 introduces the main concepts of dynamic algorithms. Chapter 4 reviews the color refinement procedure and discusses properties that are relevant for dynamic updates. Chapter 5 presents the algorithms studied in this work together with proofs of correctness and complexity analysis. Chapter 6 contains the experimental evaluation on random trees and Erdős-Rényi graphs. Finally, Chapter 7 summarizes the results and discusses possible directions for future work.

2 Notation

In this thesis, the following notions and conventions are used:

1. Natural numbers start from 1.
2. A graph G is a pair (V, E), where V is the set of vertices and $E \subseteq V \times V$ is the set of edges, consisting of unordered pairs of distinct vertices. Throughout this thesis, all graphs are simple and undirected; in particular, $(u, v) = (v, u)$ and no multiple edges are allowed.
3. For a vertex $v \in V$, the **open neighborhood** of v is:

   $$N(v) = \{u \mid \exists\, (u, v) \in E\}.$$

   The **closed neighborhood** of v is $N[v] = \{v\} \cup N(v)$.
4. For vertices $v_0, \ldots, v_t$, define

   $$N(v_0, \ldots, v_t) = \cup_{i=0}^{t} N(v_i),$$

   and similarly for $N[v_0, \ldots, v_t]$.
5. The **distance** $\mathrm{dist} : V \times V \to \mathbb{N} \cup \{0, \infty\}$ between two vertices is the length of a shortest path between them. If no such path exists, the distance is ∞.
6. The set of vertices at distance at most k from a vertex v is denoted $N_k(v)$:

   $$N_k(v) = \{u \mid \mathrm{dist}(u, v) \leq k\}.$$

7. For a subset $S \subseteq V$, the **induced subgraph** $G[S]$ is the graph with vertex set S and edge set

   $$\{(u, v) \in E \mid u, v \in S\}.$$

   For $k \in \mathbb{N}$, define the k-neighborhood subgraph of v as

   $$G_k[v] = G[N_k[v]].$$

8. A **multiset** $\{\!\{\cdot\}\!\}$ is a set such that the elements can repeat themselves.

3 Dynamic graph algorithms

Unlike static algorithms, which compute a solution for a fixed input from scratch, dynamic algorithms maintain a solution to a problem as the underlying input undergoes modifications. In this chapter, we introduce the fundamental concepts related to dynamic problems within the context of this thesis, following the definitions in [17] and [18].

Definition 1 (Dynamic Algorithm)**.** A dynamic algorithm is a data structure that maintains the solution to a problem P over an input I that evolves through a sequence of updates

$$\sigma = (u_1, u_2, \ldots, u_k)$$

. After each update u_i, the algorithm updates its internal state to represent the solution for P on the modified input I_i.

In this thesis, we focus on the case where the input I is a graph. Dynamic graph algorithms operate on graphs that evolve over time through a sequence of structural changes.

Definition 2 (Dynamic Graph)**.** A dynamic graph is a sequence of graphs

$$G_0, G_1, G_2, \ldots$$

over a fixed vertex set V, where each graph G_{i+1} is obtained from G_i by applying a single update operation, such as the insertion or deletion of an edge.

A dynamic graph algorithm is a dynamic algorithm in which the input is a graph and the updates consist of structural modifications. Such algorithms typically support two types of operations:

Definition 3 (Dynamic Graph Algorithm)**.** A dynamic graph algorithm is a dynamic algorithm in where the input is a graph and the updates consist of structural modifications to the graph.
Such an algorithm supports two types of operations:

- **update operations**, which modify the graph structure (e.g., insert(u, v) or delete(u, v) for $u, v \in V$),
- **query operations**, which return information about a property of the current graph G_i (e.g., connectivity, reachability, or color refinement classes).

A **fully dynamic graph algorithm** supports both edge insertions and deletions.

To evaluate the efficiency of a dynamic algorithm, we consider several standard complexity measures. Let $n = |V|$ and $m = |E|$. The performance of a dynamic algorithm is characterized by:

- **preprocessing time:** the time required to initialize the data structure on the initial graph G_0,
- **update time:** the time required to modify the data structure after a single update u_i,

- **query time:** the time required to answer a single query operation.

A natural baseline for efficiency is the performance of a static algorithm. Recomputing a solution from scratch after every update gives an update time of $O(T(m))$, where $T(m)$ is the running time of the best-known static algorithm for the problem. The main goal of a dynamic algorithm is to achieve an update time of $o(T(m))$, i.e., significantly faster than recomputation.

The update time of a dynamic algorithm can be measured either in the **worst case** (the maximum time required for any single update u_i) or in the **amortized sense** (the average time per update over a worst-case sequence of operations σ).

An important aspect of dynamic algorithms is the locality of updates. In many applications, a small modification to the graph affects only a limited portion of the maintained structure. Efficient algorithms aim to update only the affected parts to avoid redundant recomputation.

In this thesis, we study dynamic algorithms in the context of color refinement, focusing on how local changes in the graph propagate through the refinement process and how to bound the resulting update costs.

4 Color refinement and its properties

Color Refinement (CR) is an iterative algorithm that groups vertices according to their structural position in a graph. The process begins with an initial coloring of the vertices. In each subsequent iteration, a vertex receives a new color based on two components:

1. its color in the previous iteration, and
2. the multiset of colors of its neighbors in the previous iteration.

The procedure stops when in two consecutive iterations the number of different colors stays the same.

In this chapter, we examine the behavior of color refinement and analyze how the resulting partition changes when the graph is modified. In particular, we study how adding or removing edges - while keeping the vertex set fixed - affects the colors assigned to vertices and the resulting color classes. For simplicity, we focus on undirected graphs; however, the reasoning and results extend naturally to directed graphs as well.

4.1 Definitions and properties

Definition 4. Color Refinement (CR) [19]
Let $\lambda : V(G) \to \mathcal{C}$ be a coloring of the vertices of a graph G, where $\mathcal{C}$ is some set of colors. The coloring computed by color refinement on input (G, λ) is defined recursively: we set $\chi_G^0 := \lambda$, i.e., the initial coloring is λ. For $i \in \mathbb{N}$, the coloring χ_G^i computed by color refinement after i iterations on G is defined as

$$\chi_G^i(v) := \left(\chi_G^{i-1}(v), \{\!\{\chi_G^{i-1}(w) \mid w \in N(v)\}\!\}\right). \tag{4.1}$$

The process stops when two consecutive iterations induce the same partition. For brevity, we write $\chi^i(v)$ when the graph is clear from context.

Remark. This formulation allows an arbitrary initial coloring λ. Throughout this thesis, however, we assume a uniform initial coloring in which all vertices receive the same color (typically 1).

Remark. Each color in iteration i is a pair consisting of the previous color and the multiset of the neighbors' colors. Although this tuple representation is useful for theoretical arguments, we later represent colors using natural numbers.

Before analyzing the properties of the algorithm, we formalize the terminology that captures the structure of the intermediate colorings.

Definition 5 (Color Tuple). For a vertex v and iteration i, the value $\chi^i(v)$ defined in (4.1) is called the *color tuple* of v in iteration i.

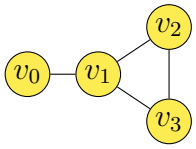
Definition 6 (Color Class). For an iteration i, a *color class* of a vertex v is a set of vertices receiving the same color tuple:

$$C^i(v) = \{\, u \in V \mid \chi^i(u) = \chi^i(v) \,\}.$$

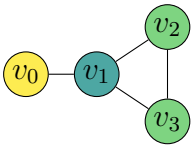
By analogy with isomorphism theory, we may also refer to a color class as an *orbit*.

Definition 7 (Asymmetric and Symmetric Vertices)**.** A vertex v is called *asymmetric* in iteration i if its color tuple $\chi^i(v)$ is unique in the graph. Otherwise, v is called *symmetric*.

To illustrate the work of the algorithm, let us consider the following example.



(a) Initial iteration ($i = 0$): initial, uniform coloring.



(b) Iterations $i = 1, i = 2$: the process stabilizes.

$$v = \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix} \quad \Bigg| \quad \chi^0 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \; \chi^1 = \begin{bmatrix} (1, \{\!\{1\}\!\}) \\ (1, \{\!\{1,1,1\}\!\}) \\ (1, \{\!\{1,1\}\!\}) \\ (1, \{\!\{1,1\}\!\}) \end{bmatrix}$$

(c) Vectors χ with color tuples of vertices $\{v_0, ..., v_3\}$ for iterations $i = 0, 1$.

Figure 4.1: Visualization of the color refinement algorithm. In iteration 1 all vertices have the same color, and hence form a single color class. In iteration 2, vertices v_0 and v_1 have unique color tuples, and are thus asymmetric. On the contrary, vertices v_2 and v_3 share the same color tuple and form a color class.

After introducing the formal definition of color refinement, we now analyze how the algorithm updates vertex colors from one iteration to the next. Although all vertices are updated simultaneously, the update rule itself is local. At each iteration, a vertex gathers the colors from its neighbors, combines this information together with its own previous color, and uses this combination to determine its new color. The following theorem formalizes this locality by showing that the color of a vertex after i iterations depends only on its radius-i neighborhood.

Theorem 4.1. *Let $G = (V, E)$ be a graph and let $v \in V$. Then the color of v in iteration i of CR is determined by the induced subgraph $G_i[v]$, consisting of all vertices within distance at most i from v and the edges between them.*

Proof. We proceed by induction on i.

1. **Base case:** Let $i = 1$, and let v be a vertex with degree $d = \deg(v)$, and neighbors $u_1, \ldots, u_d \in N(v)$. Then the updated color in iteration 1 is

$$\chi^1(v) = \left(\chi^0(v), \{\!\{\chi^0(u_1), \ldots, \chi^0(u_d)\}\!\}\right).$$

   Since the update uses only the initial color of v and the multiset of the initial colors of its neighbors, it follows that $\chi^1(v)$ depends only on the structure and vertex colors in the 1-neighborhood of v, i.e., on $G_1[v]$.

2. **Inductive step:** Assume that for all $v \in V$, the color $\chi^i(v)$ is determined only by the subgraph $G_i[v]$.

At iteration $i+1$, we compute:

$$\chi^{i+1}(v) = \left(\chi^i(v), \{\!\{\chi^i(u) | u \in N(v)\}\!\}\right).$$

By the inductive hypothesis, each $\chi^i(u)$ is determined by $G_i[u]$. Since $u \in N(v)$, any vertex in $G_{i+1}[u]$ lies within distance at most $i+1$ from v. Therefore, the information needed to compute $\chi^{i+1}(v)$ is contained entirely within $G_{i+1}[v]$.

Hence, $\chi^{i+1}(v)$ is determined by $G_{i+1}[v]$, completing the induction.

□

Thus, in each iteration of color refinement, the color tuple of a vertex depends only on its neighborhood. This locality is particularly relevant in the dynamic setting, where the graph is modified over time and the goal is to update the coloring efficiently after each change. In the early rounds, the relevant neighborhood has small radius, but its size depends strongly on the structure of the graph: in graphs with small diameter, it may already cover a large fraction of the vertices. For example, in a star graph, information from any edge modification reaches all vertices within two steps. In contrast, in graphs with large diameter or weak connectivity, the effects of a modification remain localized for many iterations. This includes, for example, path-like graphs or updates confined to a single connected component of a disconnected graph.

This observation suggests using local update strategies: after a graph update, it is often sufficient to recompute the color tuples only for vertices whose neighborhoods are affected, while reusing the coloring elsewhere.

This leads to the following question: after updating the graph, which vertex colors - and consequently which color classes - can change? In this thesis, we focus on updates that modify only the edge set while keeping the vertex set fixed; in particular, an update may add or remove one or more edges.

Theorem 4.2. *Let $G = (V, E)$ be a graph with CR colorings $\chi^0, \chi^1, \ldots, \chi^k$, where colors are understood as color tuples. Let $G' = (V, E')$ be a graph obtained by modifying the edge set. Let $S \subseteq V$ be the set of vertices incident to modified edges, defined as:*

$$S = \{v \in V \mid \exists\, e \in E \triangle E' \text{ such that } v \in e\}.$$

Run CR on G', resulting in colorings $\chi'^0, \chi'^1, \ldots, \chi'^l$. Then, for each iteration $i \leq \min(k, l)$, only the vertices in $N_i[S]$ may have changed their color tuple. Consequently, the color tuples of all vertices in $V \setminus N_i[S]$ remain unchanged.

Proof. Let $v \in N_i[S]$. Then there exists $u \in S$ such that $\mathrm{dist}(v, u) \leq i$, so $u \in G_i[v]$. Since $u \in S$, it has changed its set of neighbors, so the local subgraph $G_i[v]$ may differ between G and G'. Therefore, the color tuple at step i for vertex v may change, and so $\chi^i(v) \neq \chi'^i(v)$ is possible.

Conversely, let $v \notin N_i[S]$. Then for all $u \in S$, $\mathrm{dist}(v, u) > i$, which implies that no vertex in $G_i[v]$ was affected by the edge change. Thus, $G_i[v]$ is the same in both G and G', and the CR computation proceeds identically. Therefore, $\chi^i(v) = \chi'^i(v)$. □

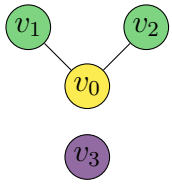
Corollary 4.2.1. If two vertices $v_1, v_2 \in V \setminus N_i[S]$ satisfy $\chi^i(v_1) = \chi^i(v_2)$, then they will remain in the same color class in G' at step i:

$$\chi'^i(v_1) = \chi'^i(v_2).$$

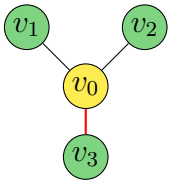
That is, unaffected vertices that had the same color in iteration i will retain the same color after the change, even if their color class itself changes.

Remark. The theorem states that vertices in $N_i[S]$ *may* change their color tuple - but this is not guaranteed. This illustrates the following example.

Let us consider a graph G with $V = \{v_0, v_1, v_2, v_3\}$ and $E = \{(v_0, v_1), (v_0, v_2)\}$ (v_3 is an isolated vertex). Let (v_0, v_3) be a new edge.



(a) Graph G with edges $(v_0, v_1), (v_0, v_2)$.



(b) Graph G with new edge (v_0, v_3) (red).

Figure 4.2: The graph G and the modified graph G' obtained by adding one edge. The colors of vertices correspond with the final CR partition.

The vertex v_0 is affected by the change already in the first iteration, since it is incident to the new edge. Its color tuples are:

$$\chi^1(v_0) = (1, \{\!\{1, 1\}\!\}), \quad \chi'^1(v_0) = (1, \{\!\{1, 1, 1\}\!\}).$$

In both cases, these color tuples are unique in the graph ($C^1(v_0) = \{v_0\} = C'^1(v_0)$). Hence, even though the neighborhood of v_0 changed, its color class remained the same. In other words, being contained in $N_1[S]$ is only a necessary condition for a vertex to be affected, not a sufficient one.

A key question in this thesis is whether we can efficiently update the result of the color refinement algorithm after making a small change to the graph, without recomputing the entire coloring from scratch.

The locality of the process described above allows us to avoid running the full refinement procedure. Instead, it is sufficient to update only the region affected by the modification. In this way, the coloring can be maintained more efficiently as the graph evolves.

Before introducing the first update algorithm, we formalize the color refinement process using integer labels. As before, we assume that the initial coloring is uniform.

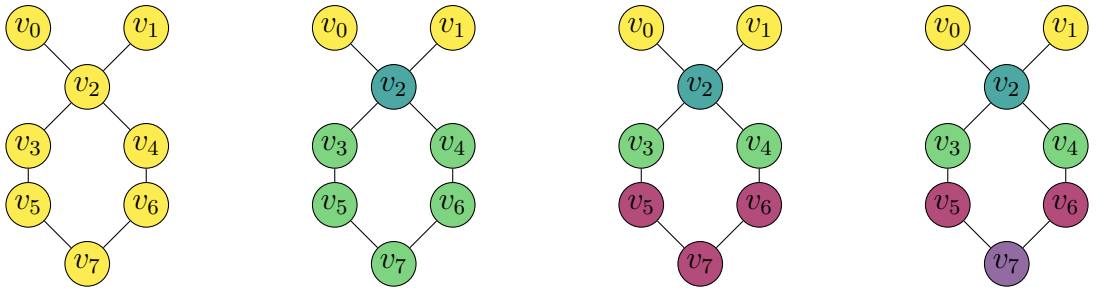
Definition 8 (Color Refinement with Integer Labels)**.** Let $G = (V, E)$ be a graph with fixed vertex labels: $V = (v_0, \ldots, v_{n-1})$. The color refinement process with integer labels is defined as a sequence of vectors *(colorings)* $c^0, c^1, \ldots, c^k \in \mathbb{N}^n$, where $c^0 = \mathbf{1}$ is the all-ones vector. Given c^i, the next coloring c^{i+1} is computed as follows:

1. For each vertex $v \in V$, compute its color tuple:

$$\gamma^{i+1}(v) = \left(c^i(v), \{\!\{c^i(w) \mid w \in N(v)\}\!\}\right).$$

2. Initialize the first *color* (*label/signature*), $c^{i+1}(v_0) := 1$. Then, for $j = 1, \ldots, n-1$, set:
   - If there exists $\ell < j$ such that $\gamma^{i+1}(v_j) = \gamma^{i+1}(v_\ell)$, then $c^{i+1}(v_j) := c^{i+1}(v_\ell)$.
   - Otherwise, assign $c^{i+1}(v_j)$ the smallest natural number not yet used.
3. Stop if the number of colors in iteration i equals the number of colors in iteration $i + 1$.

The integers in c^i serve as labels for the color classes defined by the tuples γ^i. To visualize this process let us look at the example below.



(a) Initial coloring **(b)** Iteration 1 **(c)** Iteration 2 **(d)** Iterations 3, 4

$$v = \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \end{bmatrix} \quad \Bigg| \quad c^0 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \; c^1 = \begin{bmatrix} 1 \\ 1 \\ 2 \\ 3 \\ 3 \\ 3 \\ 3 \\ 3 \end{bmatrix}, \; c^2 = \begin{bmatrix} 1 \\ 1 \\ 2 \\ 3 \\ 3 \\ 4 \\ 4 \\ 4 \end{bmatrix}, \; c^3 = c^4 = \begin{bmatrix} 1 \\ 1 \\ 2 \\ 3 \\ 3 \\ 4 \\ 4 \\ 5 \end{bmatrix}$$

(e) The sequence of colorings c.

Figure 4.3: Example of color refinement with integer labels. Colors of vertices correspond to labels assigned in each coloring. The process stabilizes after iteration 3 (so iterations 3 and 4 form the same partition).

The refinement process for the graph in Figure 4.3 proceeds as follows. In the first iteration, the vertices are distinguished by degree, which results in three distinct color tuples:

$$\begin{aligned} \chi^1(v_0) = \chi^1(v_1) &= (1, \{\!\{1\}\!\}), \\ \chi^1(v_2) &= (1, \{\!\{1,1,1,1\}\!\}), \\ \chi^1(v_3) = \cdots = \chi^1(v_7) &= (1, \{\!\{1,1\}\!\}). \end{aligned}$$

Based on these tuples, the vertices receive integer labels from 1 to 3. Using these new labels, the second iteration produces the following color tuples:

$$\begin{aligned} \chi^2(v_0) = \chi^2(v_1) &= (1, \{\!\{2\}\!\}), \\ \chi^2(v_2) &= (2, \{\!\{1,1,3,3\}\!\}), \\ \chi^2(v_3) = \chi^2(v_4) &= (3, \{\!\{2,3\}\!\}), \\ \chi^2(v_5) = \cdots = \chi^2(v_7) &= (3, \{\!\{3,3\}\!\}). \end{aligned}$$

Finally, in iteration 3 the color tuples of vertices $v_0, ..., v_4$ reamin the same as before, and the other tuples are:

$$\begin{aligned} \chi^3(v_5) = \chi^3(v_6) &= (4, \{\!\{3,4\}\!\}), \\ \chi^3(v_7) &= (4, \{\!\{4,4\}\!\}). \end{aligned}$$

Iteration 4 gives the same coloring and hence the process ends.

4.2 Complexity

The running time of the color refinement (CR) algorithm depends on how the color tuples are represented and how new colors are assigned in each iteration.

Naive implementation

A direct implementation follows Definition 4 by explicitly constructing, for each vertex v, the multiset of colors of its neighbors and comparing these multisets. This typically requires sorting the neighbor colors. For a vertex v, this takes $O(\deg(v) \log \deg(v))$ time. Summed over all vertices, one iteration costs

$$O\left(\sum_{v \in V} \deg(v) \log \deg(v)\right) = O(m \log n).$$

Since there can be at most $n - 1$ refinement rounds, this yields a worst-case running time of $O(nm \log n)$.

Labeling color tuples

More efficient implementations avoid explicit sorting of multisets. Instead, color tuples are encoded and mapped to integer labels using hashing or global sorting over all vertices, as described by [20]. In this approach, the multiset of neighbor colors is aggregated in linear time, and equal tuples are identified through a dictionary mapping. This allows computing one refinement round in $O(m)$ time.

Partition refinement algorithms

More efficient algorithms compute the stable coloring directly using partition refinement techniques. These methods achieve a running time of $O(m \log n)$. The historical development of these algorithms is outlined in the introduction of [21].

Implementation in this thesis

In the main part of this thesis, we describe the algorithms using explicit color tuples, which leads to the $O(\deg(v) \log \deg(v))$ cost per vertex in tuple computation. However, the dynamic update procedures are independent of the specific representation of color tuples. In particular, the hash-based labeling described in Appendix can be used to reduce the cost of tuple computation to $O(\deg(v))$ per vertex.

5 Algorithms

5.1 Problem definition

Let $G = (V, E)$ be an initial graph, and let $(\chi_G^0, \chi_G^1, \ldots, \chi_G^k)$ be the sequence of colorings produced by color refinement on G, where χ_G^k is stable. Let $\sigma = (u_1, u_2, \ldots, u_s)$ be a sequence of edge insertions and deletions, and let $G' = (V, E')$ be the graph obtained from G after applying all updates in σ.

Furthermore, let $(\chi_{G'}^0, \chi_{G'}^1, \ldots, \chi_{G'}^l)$ be the sequence of colorings produced by the static color refinement algorithm on G', where $\chi_{G'}^l$ is stable.

Definition 9 (Equivalent Colorings)**.** Two colorings $\chi, \chi^* : V \to \mathcal{C}$ are called equivalent if they induce the same partition of V. That is, for all $u, v \in V$,

$$\chi(u) = \chi(v) \iff \chi^*(u) = \chi^*(v).$$

Definition 10 (Dynamic Color Refinement Problem)**.** Given the coloring information produced by color refinement on the original graph G, maintain or compute a coloring

$$\chi^* : V \to \mathcal{C}$$

for the updated graph G' such that χ^* is equivalent to the stable coloring $\chi_{G'}^l$ produced by static color refinement on G'.

An algorithm is correct if, after each update, it maintains a coloring that coincides with the stable coloring obtained by applying color refinement to the current graph.

To update a color refinement coloring after an edge modification, we do not recompute the coloring from scratch. Instead, we start from the coloring of the original graph and update it so that it matches the result of color refinement on the modified graph.

After a change, only some vertices may need to be updated. The algorithms in this chapter identify a set of vertices whose colors may have changed and update them iteratively until the coloring is stable again. The algorithms differ in how this set is determined.

We consider three update algorithms:

- CUP: *Color Update Propagation*, which processes all vertices in the region that may be affected by the modification;
- CAS: an extension of CUP using *asymmetric vertex pruning*, which skips vertices whose colors are already uniquely determined;
- COR: a variant based on *orbit/class pruning*, which skips entire color classes that remain unchanged.

The following sections describe and compare these algorithms in detail.

5.2 Color Update Propagation (CUP)

The first update strategy we consider is CUP, the simplest and most direct method for applying the locality principle. Given a graph update (edge insertion or deletion), the set of affected endpoints, and the refinement history of the original graph, CUP recomputes color tuples only for these initially affected vertices and then propagates the update outward through their neighborhoods. In each iteration, the algorithm compares the newly computed tuples with those from the original refinement, adjusts the corresponding color classes, and adds newly affected vertices to the queue. The process continues as long as changes propagate and stops once the refinement stabilizes.

Using the information from the original refinement, CUP effectively reconstructs the CR process on the updated graph - but only in the parts that potentially require modification. If the updated graph needs more refinement rounds than the original one, CUP continues with standard color refinement.

The procedure is effective but conservative: it propagates through the entire region that might be influenced by the modification, even when some vertices could already be excluded. The next variants, CAS and COR, reduce this unnecessary propagation.

To distinguish data before and after the graph modification, we use the subscripts "old" and "upd". In the algorithm, the refinement history of the original graph is stored in the object *old_iterations*, defined as

$$\mathrm{old_iterations}[i] = \big(c^i_{\mathrm{old}},\ \gamma^i_{\mathrm{old}},\ D^i_{\mathrm{old}}\big). \tag{5.1}$$

For each $i \in \{0, \ldots, k-1\}$, c^i_{old} is the coloring from iteration i, γ^i_{old} is the corresponding array of color tuples, and D^i_{old} is the dictionary representing the partition at iteration i as a mapping from color tuples to their integer labels and color classes:

$$D^i : \gamma \longmapsto \big(c(\gamma),\, C(\gamma)\big),$$

where $c(\gamma) \in \mathbb{N}$ is the integer label assigned to the tuple γ, and $C(\gamma) \subseteq V$ is the corresponding color class, i.e.,

$$C(\gamma) = \{\, v \in V \mid \gamma^i(v) = \gamma \,\}.$$

Finally, we denote by S the set of vertices directly affected by the modification:

$$S = \big\{\, v \in V \mid \exists\, e \in E \triangle E' \text{ such that } v \in e \,\big\}.$$

5.2.1 Pseudocode

Algorithm 1 CUP

Input: Updated graph G_{upd}; affected vertices S; refinement history of original graph
Output: Updated colorings c^i_{upd} for G_{upd}

1: Initialize list c_{upd} with uniform coloring
2: $q \leftarrow S$
3: $m_{pre} \leftarrow -1$
4: $m_{now} \leftarrow 1$
5: $i \leftarrow 1$
6: $k \leftarrow$ len(old_iterations)
7: $c^0_{\text{upd}} \leftarrow \mathbf{1}$
8: **while** $i < k$ **and** $m_{\text{pre}} \neq m_{\text{now}}$ **and** $q \neq \emptyset$ **do**
9: $\quad \gamma^i_{\text{upd}} \leftarrow$ UPDATE_COLOR_TUPLES$(G_{\text{upd}},\ q,\ c^{i-1}_{\text{upd}},\ \gamma^i_{\text{old}})$
10: $\quad c^i_{\text{upd}},\ D^i_{\text{upd}} \leftarrow$ UPDATE_COLORING$(G_{\text{upd}},\ q,\ \gamma^i_{\text{upd}}, \text{old_iterations}[i])$
11: $\quad q \leftarrow$ UPDATE_QUEUE$(G_{\text{upd}},\ q)$
12: $\quad$ append c^i_{upd} to c_{upd}
13: $\quad m_{pre} \leftarrow m_{now}$
14: $\quad m_{now} \leftarrow |D^i_{\text{upd}}|$
15: **end while**
16: **if** $q = \emptyset$ **and** $m_{\text{pre}} \neq m_{\text{now}}$ **then** ▷ local update finished, refinement incomplete
17: $\quad$ **for** $t \leftarrow i+1, \ldots, k$ **do**
18: $\quad\quad$ append c^t_{old} to c_{old}
19: $\quad$ **end for**
20: **else if** $m_{\text{pre}} \neq m_{\text{now}}$ **then** ▷ refinement incomplete, queue non-empty
21: $\quad$ continue with standard CR on G_{upd}
22: **end if**
23: **return** c_{upd}

Algorithm 2 UPDATE_COLOR_TUPLES

Input: $G_{\text{upd}},\ q,\ c^{i-1}_{\text{upd}},\ \gamma^i_{\text{old}}$
Output: γ^i_{upd}

1: $\gamma^i_{\text{upd}} \leftarrow \gamma^i_{\text{old}}$
2: **for** each $v \in q$ **do**
3: $\quad \gamma^i_{\text{upd}}(v) \leftarrow \left(c^{i-1}_{\text{upd}}(v), \{\!\{c^{i-1}_{\text{upd}}(w) \mid w \in N_{\text{upd}}(v)\}\!\}\right)$
4: **end for**
5: **return** γ^i_{upd}

Remark. The pseudocode defines a color tuple as a multiset of neighbor colors, consistent with the theoretical formulation in Section 4. In the actual implementation, however, we use the hashing-based method of [22]: tuple signatures are computed via sparse matrix-vector multiplication using the adjacency matrix. This avoids constructing multisets explicitly and reduces the memory required to store each iteration, while preserving the refinement information. The asymptotic running time remains unchanged; only constant factors differ.

Remark. In line 11 of Algorithm 3, a new color is assigned to an updated vertex. Using a counter is one possible way to do it. With this approach, the final integer labels may differ from those obtained by running CR from scratch, but the induced vertex partition remains identical.

Algorithm 3 UPDATE_COLORING

Input: $G_{\text{upd}},\ q,\ \gamma^i_{\text{upd}},\ \text{old_iterations}[i]$
Output: $c^i_{\text{upd}}, D^i_{\text{upd}}$

1: $c^i_{\text{upd}} \leftarrow c^i_{\text{old}}$
2: $D^i_{\text{upd}} \leftarrow D^i_{\text{old}}$
3: $\text{free_color} \leftarrow |D^i_{\text{upd}}| + 1$
4: **for** each $v \in q$ **do**
5: Remove v from its class in $D^i_{\text{upd}}[\gamma^i_{\text{old}}(v)]$
6: **if** class empty **then**
7: Remove tuple from the dictionary
8: **end if**
9: **if** $\gamma^i_{\text{upd}}(v) \in D^i_{\text{upd}}$ **then**
10: $c^i_{\text{upd}} \leftarrow$ color in $D^i_{\text{upd}}[\gamma^i_{\text{upd}}(v)]$
11: **else**
12: $c^i_{\text{upd}}(v) \leftarrow \text{free_color}$
13: $D^i_{\text{upd}}[\gamma^i_{\text{upd}}(v)] \leftarrow (c^i_{\text{upd}}(v), \{v\})$
14: $\text{free_color} \leftarrow \text{free_color} + 1$
15: **end if**
16: **end for**
17: **return** $c^i_{\text{upd}}, D^i_{\text{upd}}$

Algorithm 4 UPDATE_QUEUE

Input: $G_{\text{upd}},\ q$
Output: updated queue q_{new}

1: $q_{\text{new}} \leftarrow \emptyset$
2: **for all** $v \in q$ **do**
3: $q_{\text{new}} \leftarrow q_{\text{new}} \cup N_{\text{upd}}[v]$
4: **end for**
5: **return** q_{new}

Remark. In line 16 of Algorithm 1, if the queue q becomes empty, no further corrections are needed, and the algorithm simply appends the remaining colorings from the original refinement history. For CUP, the queue becomes empty only when the initial set S is empty; however, this condition plays an important role in the CAS and COR variants.

5.2.2 Correctness

Theorem 5.1 (Correctness of CUP). *Let $G_{\text{old}} = (V, E)$ be a graph, and let $G_{\text{upd}} = (V, E')$ be obtained by modifying the edge set. Let $S \subseteq V$ be the set of vertices incident to modified edges. Assume uniform initial coloring in both runs.*

Let $\left(c^i_{\text{upd}}\right)_{i\geq 0}$ be the sequence of colorings produced by Algorithm 1 on input $(G_{\text{upd}}, S, \text{old_iterations})$. Let $\left(c^i_{\text{cr}}\right)_{i\geq 0}$ be the colorings obtained by running standard color refinement from scratch on G_{upd}.

Then, for every iteration i that Algorithm 1 completes, the coloring c^i_{upd} induces the same vertex partition as c^i_{cr}. Consequently, Algorithm 1 produces a coloring equivalent to the stable coloring of G_{upd}, and thus solves the dynamic color refinement problem (Definition 10).

Proof. We prove by induction on $i \geq 0$ the following points. After completing iteration i of Algorithm 1:

(I1) The dictionary D^i_{upd} contains exactly the color tuples $\gamma^i_{\text{cr}}(v)$ that are in the from-scratch run, and each key maps to precisely the corresponding color class.

(I2) The coloring c^i_{upd} is consistent with D^i_{upd} and therefore induces the same partition as c^i_{cr}.

(I3) The queue at the beginning of iteration $i+1$ equals $N_i[S]$.

Base case ($i = 0$). Both algorithms start with uniform coloring, so (I2) holds. By definition, $q = S = N_0[S]$, so (I3) holds.

Inductive step. By Theorem 4.2, the only vertices whose color tuples may differ from the original refinement in iteration i are those in $N_i[S]$. By the induction hypothesis (I3), the algorithm starts iteration i with $q = N_{i-1}[S]$ and then updates the queue to $q \leftarrow N_{\text{upd}}[q] = N_i[S]$.

Thus, in UPDATE_COLOR_TUPLES (Algorithm 2) we only need to recompute $\gamma^i_{\text{upd}}(v)$ for vertices $v \in q = N_i[S]$. For all vertices $v \notin q$ we keep $\gamma^i_{\text{upd}}(v) = \gamma^i_{\text{old}}(v)$. By Theorem 4.2, these unchanged tuples matches the $\gamma^i_{\text{cr}}(v)$ from the from-scratch refinement. Consequently, the entire assignment γ^i_{upd} is the same as the tuple assignment of the from-scratch run in iteration i.

In UPDATE_COLORING (Algorithm 3) we start from D^i_{old} and
(i) remove each processed vertex $v \in q$ from the class indexed by its old tuple,
(ii) insert it into the class indexed by its updated tuple, creating a new class if necessary.

Since γ^i_{upd} matches the from-scratch tuple assignment, the resulting dictionary D^i_{upd} matches exactly the partition produced by the from-scratch run in iteration i, proving (I1). Because c^i_{upd} is updated consistently with D^i_{upd}, property (I2) follows.

Finally, Algorithm 4 sets $q \leftarrow N_{\text{upd}}[q] = N_i[S]$ for the next iteration, showing (I3).

Thus (I1)-(I3) hold for every completed iteration i, and in particular c^i_{upd} induces the same partition as c^i_{cr}.

Since c^i_{upd} coincides with c^i_{cr} in every iteration, the final coloring is equivalent to the stable coloring of G_{upd}. By Definition 10, this shows that Algorithm 1 correctly solves the dynamic color refinement problem. □

5.2.3 Complexity

Time complexity

Let $n = |V|$ and $m = |E|$. Let t denote the number of refinement iterations of color refinement on the original graph G, i.e., $k = \text{len}(\textit{old_iterations})$. For each iteration i, let q_i be the current queue. We assume that all dictionary operations take expected $O(1)$ time.

The running time of one iteration of CUP is determined by the three update procedures.

For each vertex $v \in q_i$, the algorithm recomputes the color tuple

$$\gamma^i_{\text{upd}}(v) = \left(c^{i-1}_{\text{upd}}(v), \{\!\{\, c^{i-1}_{\text{upd}}(w) \mid w \in N_{\text{upd}}(v) \,\}\!\}\right).$$

The cost of computing $\gamma^i_{\text{upd}}(v)$ depends on how color tuples are represented.

In the explicit tuple-based implementation, the multiset of neighbor colors is constructed and sorted. Thus, for one vertex v, this requires

$$O(\deg(v) \log \deg(v)).$$

Alternatively, using the hash-based encoding described in the Appendix, the tuple computation can be performed by aggregating the neighbor colors once, which requires

$$O(\deg(v)).$$

Summing over all vertices in q_i gives

$$T^i_{\text{tuples}} = \begin{cases} O\Big(\sum_{v\in q_i} \deg(v)\log\deg(v)\Big) & \text{(tuple-based)} \\ O\Big(\sum_{v\in q_i} \deg(v)\Big) & \text{(hash-based)} \end{cases}$$

2. Function UPDATE_COLORING:
For each vertex $v \in q_i$, we perform:
- one deletion from a color class
- one hash lookup for the new tuple
- one insertion into a class or creation of a new class

All those operations, as operations on a hash map, cost:

$$T^i_{coloring} = O(|q_i|)$$

3. Function UPDATE_QUEUE:
This function returns $q_{i+1} = N[q_i]$. To construct such set, we need to again process the neighbors of each vertex in the queue, which requires

$$T^i_{queue} = O\big(\sum_{v\in q_i} deg(v)\big)$$

4. Cost per iteration:
Combining the three parts, the cost of one iteration is

$$T^i_{\text{CUP}} = O\left(\sum_{v\in q_i} \deg(v)\log\deg(v) + |q_i|\right).$$

Since $|q_i| \le \sum_{v\in q_i}(\deg(v)+1)$, the tuple computation dominates, and we obtain

$$T^i_{\text{CUP}} = \begin{cases} O\Big(\sum_{v\in q_i} \deg(v)\log\deg(v) + |q_i|\Big) & \text{(tuple-based)} \\ O\Big(\sum_{v\in q_i} \deg(v) + |q_i|\Big) & \text{(hash-based)} \end{cases}$$

5. Total cost of CUP algorithm:
Summing over all iterations gives

$$T_{\text{CUP}} = \begin{cases} O\Big(\sum_{i=1}^{t}\sum_{v\in q_i} \deg(v)\log\deg(v)\Big) & \text{(tuple-based)} \\ O\Big(\sum_{i=1}^{t}\sum_{v\in q_i} \deg(v)\Big) & \text{(hash-based)} \end{cases}$$

Worst-case bound

In the worst case, the queue contains all vertices in every iteration, i.e., $q_i = V$. Then one iteration costs

$$O\left(\sum_{v \in V} \deg(v) \log \deg(v)\right) = O(m \log n).$$

Since $k \leq n - 1$ for color refinement [19], the total running time is bounded by

$$\begin{cases} O(nm \log n) & \text{(tuple-based)} \\ O(nm) & \text{(hash-based)} \end{cases}$$

Memory Complexity

During execution, the algorithm maintains:

1. the graph representation, requiring $O(n + m)$ space;
2. the queue q_i, requiring at most $O(n)$ space;
3. the current coloring c^i_{upd}, requiring $O(n)$ space;
4. the tuple array γ^i_{upd}, requiring $O(m)$ space in the explicit representation;
5. the dictionary D^i_{upd}, requiring $O(n)$ space.

In addition, the refinement history of the original graph is stored in *old_iterations*. If the original refinement requires k iterations, storing all colorings, tuples, and dictionaries requires

$$O(k(n + m))$$

space in the tuple-based implementation.

In the hash-based implementation (Appendix), the tuple array is replaced by a hash array of size $O(n)$. Thus, storing the refinement history requires

$$O(kn)$$

space.

Therefore, the total memory usage is

$$\begin{cases} O(k(n + m)) & \text{(tuple-based)} \\ O(kn) & \text{(hash-based).} \end{cases}$$

In practice, the number of refinement rounds k is typically small, so the memory usage is close to linear in the size of the graph.

Comparison to static color refinement.

Recomputing the coloring from scratch using a static color refinement algorithm requires $O(m \log n)$ time [21]. In contrast, CUP processes only the vertices contained in the queues q_i, and its running time depends on the size of the affected region.

In particular, if the update affects only a small subset of vertices, the total running time

$$O\left(\sum_{i=1}^{t} \sum_{v \in q_i} \deg(v) \log \deg(v)\right)$$

(or $O\left(\sum_{i=1}^{t} \sum_{v \in q_i} \deg(v)\right)$ in the hash implementation) can be significantly smaller than $O(m \log n)$.

Motivation: positive examples

Affected vertices contained in one disconnected component.

The strongest advantage of the propagation method appears when the update is contained in a relatively small part of the graph. In this case the queue q_i grows only within the affected component and its immediate neighborhood, and never reaches the rest of the graph. As a result, the total work performed by CUP is

$$O\left(\sum_{v \in R} \deg(v) \log \deg(v)\right),$$

where $R = \bigcup_i q_i$ is the set of all vertices whose color tuples actually change during the update.

Small number of refinement rounds.

Although the pessimistic bound for the number of refinement rounds is $n - 1$ for a graph with n vertices [19], in many other graphs the refinement stabilizes very quickly, and the incremental update procedure profits from this behavior.

An example is the class of random graphs $G(n, \frac{1}{2})$. It is known that 1 -WL stabilizes after only two refinement rounds with high probability (see [3]). In this situation the number of refinement rounds can be treated as a constant, and therefore CUP performs only a constant number of update steps. This gives a worst-case running time of $O(m \log n)$.

Importantly, this bound does not mean that CUP actually scans all edges. When the update set S is small, the first update round processes only the vertices in S, and the second round considers only their neighborhoods, which may still be much smaller than the full graph. However, without any further selection of vertices (as performed by the next two queue-pruning strategies), the third round of CUP - almost surely the final "check round" in $G(n, \frac{1}{2})$ - will reach almost all vertices, since such random graphs have diameter 2 with high probability [23]. Thus plain CUP typically reaches a large part of the graph in the last round, but the earlier rounds remain very small when S is small.

In next sections we will see that using selective queue-pruning strategies (CAS and COR) can reduce the number of vertices touched in the later rounds as well.

Because only a constant number of colorings must be stored, the memory usage in this case is also $O(n + m)$.

Sparse and low-degree graphs.

If the graph has bounded maximum degree δ, then every queue update and every tuple update costs $O(|q_i|\, \delta \log \delta)$ in round i. For small δ, this gives nearly linear running time even in the worst case. Furthermore, although the worst-case number of refinement rounds is $n-1$, in many graphs color refinement stabilizes after only a few iterations. In such cases, the total running time is correspondingly reduced.

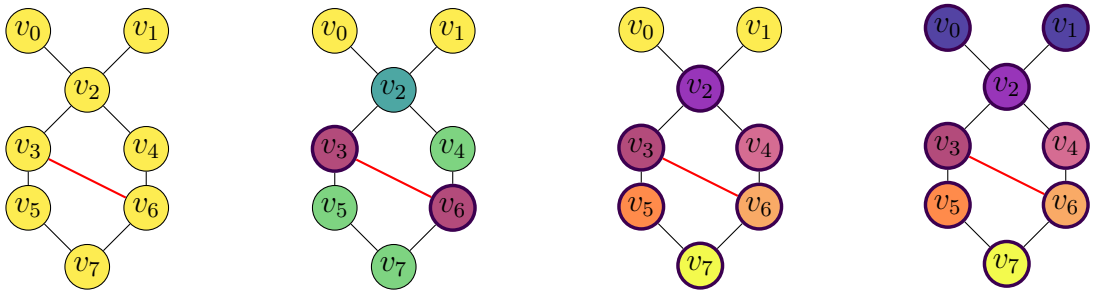
Example

To demonstrate how CUP propagates updates after a graph modification, we consider the graph from the previous example with one additional edge (v_3, v_6). The new edge affects exactly its two endpoints, so the initial queue is $q = \{v_3, v_6\}$. CUP then follows the refinement history of the original graph and updates only those vertices whose color tuples may change.

Figure 5.1 shows the evolution of the queue and the updated coloring over four iterations. In each subfigure, vertices in the current queue are highlighted with thick borders, and the colors

reflect the updated labels c^i_{upd}. Iteration 0 begins with the uniform coloring and the initial queue. In iteration 1, the vertices from the initial queue are updated and change propagates to their neighbors of v_3 and v_6, enlarging the queue. In iteration 2, propagation reaches the upper part of the graph, and all vertices enter the queue. By iteration 3, the refinement stabilizes. Even though the vertices $\{v_0, v_1\}$ do not change their color class, they are still assigned a new color.

As the example illustrates, CUP updates only the vertices reached through the local neighborhood of the modification; all other vertices preserve their color from the original refinement.



(a) Initial coloring, $q = \{v_3, v_6\}$ **(b)** Iteration 1, $q = \{v_2, ..., v_7\}$ **(c)** Iteration 2, $q = \{v_0, ..., v_7\}$ **(d)** Iteration 3

$$v = \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \end{bmatrix} \quad c^0_{upd} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \; c^1_{upd} = \begin{bmatrix} 1 \\ 1 \\ 2 \\ \cancel{3}\ 4 \\ 3 \\ 3 \\ \cancel{3}\ 4 \\ 3 \end{bmatrix}, \; c^2_{\text{upd}} = \begin{bmatrix} 1 \\ 1 \\ \cancel{2}\ 5 \\ \cancel{3}\ 6 \\ \cancel{3}\ 7 \\ \cancel{4}\ 8 \\ \cancel{4}\ 9 \\ \cancel{4}\ 10 \end{bmatrix}, \; c^3_{\text{upd}} = \begin{bmatrix} \cancel{1}\ 6 \\ \cancel{1}\ 6 \\ \cancel{2}\ 7 \\ \cancel{3}\ 8 \\ \cancel{3}\ 9 \\ \cancel{4}\ 10 \\ \cancel{4}\ 11 \\ \cancel{5}\ 12 \end{bmatrix}$$

(e) Updated colorings for iterations 0 to 3. Crossed entries mark colors that changed in that iteration; the number next to them is the new color.

Figure 5.1: Example of the CUP algorithm applied to the graph from the previous example after adding the edge (v_3, v_6) (shown in red). Vertex colors match color classes. Thick borders indicate the processed queue; each subfigure shows the queue at the end of the corresponding iteration.

5.3 CAS: pruning unchanged asymmetric vertices

The baseline CUP algorithm updates colors by processing all vertices in the neighborhood of the initial modification, with the affected region growing according to the iteration number. However, during this process we can often detect vertices that are guaranteed not to be influenced by the change.

An important observation is that asymmetric vertices are stable under color refinement. Color refinement only refines existing classes by splitting them; it never merges distinct classes. Consequently, once a vertex becomes asymmetric in some iteration, it remains asymmetric for the rest of the refinement process.

In the context of update algorithms, this fact becomes even more useful. If a vertex is asym-

metric in both the original and the updated refinement runs, then not only is the vertex itself unaffected by the graph modification, but the change also does not propagate through it to its neighbors. CAS implements this concept by removing such vertices from the processing queue, and hence reducing unnecessary work compared to CUP.

Let us now formalize this observation. The following lemma shows that if an asymmetric vertex changes its color, then the only vertices that may be affected in the next iteration are its neighbors. The subsequent theorem shows that the proposed pruning strategy is correct: vertices that remain asymmetric in both the original and the updated refinement runs can indeed be safely excluded from further processing.

Lemma 5.3.1. Let $G = (V, E)$ be a graph with CR colorings $\chi^0, \chi^1, \ldots, \chi^k$. If $v \in V$ is asymmetric at iteration $i < k$, i.e. $C^i(v) = \{v\}$, then for every $u \in N(v)$ we have

$$C^{i+1}(u) \subseteq N(v).$$

Proof. Let $u_1 \in N(v)$. If some $u_2 \neq u_1$ belongs to $C^{i+1}(u_1)$, then the multisets of neighbor colors of u_1 and u_2 at step i are equal. In particular, both must contain $\chi^i(v)$, which occurs only at v. Thus both u_1 and u_2 are adjacent to v, so $u_2 \in N(v)$. Hence $C^{i+1}(u_1) \subseteq N(v)$. □

Now consider an incremental update. If a vertex v is asymmetric in both the original and the updated refinement, then its neighbors are influenced by exactly the same type of change in both runs: a unique color is replaced by another unique color. Consequently, all tuple equalities among neighbors that held in the original run continue to hold in the updated run.

This leads to the following theorem.

Theorem 5.2. *Let $G = (V, E)$ have CR colorings $\chi^0, \ldots, \chi^k$, and let $G' = (V, E')$ be obtained by modifying edges, affecting vertices S and resulting in colorings $\chi'^0, \ldots, \chi'^j$. Suppose that for some $i < \min(k, j)$ a vertex v is asymmetric in both graphs, i.e. $C^i(v) = C'^i(v) = \{v\}$. Let $u_1, u_2 \in N(v) \setminus S$ be neighbors of v such that their neighborhoods differ only in v, i.e.*

$$\chi^i(w) = \chi'^i(w) \quad \forall\, w \in N[u_1, u_2] \setminus \{v\}.$$

Then

$$C^{i+1}(u_1) = C^{i+1}(u_2) \iff C'^{i+1}(u_1) = C'^{i+1}(u_2).$$

Proof. By Lemma 5.3.1, for each $u \in N(v)$ we have

$$C^{i+1}(u) \subseteq N(v), \qquad C'^{i+1}(u) \subseteq N'(v).$$

Thus the equality of $C^{i+1}(u_1)$ and $C^{i+1}(u_2)$ depends only on colors of vertices in $N(v)$, and likewise in G'. Since $u_1, u_2 \notin S$, we have $N(u_l) = N'(u_l)$ for $l = 1, 2$. The only possible color change in their neighborhoods is at v, whose colors $\chi^i(v)$ and $\chi'^i(v)$ are both unique. Replacing one unique color with another does not affect equality of the multisets. Hence the claim. □

As a direct algorithmic consequence of Theorem 5.2, a vertex that is asymmetric in both refinement runs cannot cause any change in its neighbors' tuples. Therefore, such a vertex may be processed without propagating the update further.

Corollary 5.2.1. In Algorithm 1, if v is unique in both the original and updated coloring at iteration i, we may assign v its original color, $c^i_{\text{upd}}(v) = c^i_{\text{old}}(v)$. In this case, while processing v, we do not need to add $N'(v)$ to the queue, since this color assignment leaves all neighbors' color tuples unchanged, and thus the resulting coloring remains a correct CR coloring.

Proof. If v is unique in both graphs, the color tuple $\chi^i(v)$ does not appear on any other vertex. Replacing $c^i_{\text{upd}}(v)$ with $c^i_{\text{old}}(v)$ therefore does not affect the multisets of neighbor colors for any $u \in N'(v)$, and no neighbor requires reprocessing. By Theorem 5.2, the CR refinement proceeds correctly without adding $N'(v)$ to the queue. □

5.3.1 Pseudocode

We now present the algorithm implementing Theorem 5.2, an alternative to UPDATE_QUEUE in Algorithm 1.

Algorithm 5 CAS (UPDATE_QUEUE_REMOVE_ASYMMETRIC_VERTICES)

Input: $G_{\text{upd}},\ q,\ \gamma^i_{\text{upd}},\ D^i_{\text{upd}},\ c^i_{\text{upd}},\ \text{old_iterations}[i]$
Output: Updated queue q_{new}, updated coloring c^i_{upd}

$q_{\text{new}} \leftarrow \emptyset$
$A \leftarrow \emptyset$ ▷ unchanged asymmetric vertices
for each vertex $v \in q$ **do**
  $C_{\text{old}} \leftarrow D^i_{\text{old}}[\gamma^i_{\text{old}}(v)]$
  $C_{\text{upd}} \leftarrow D^i_{\text{upd}}[\gamma^i_{\text{upd}}(v)]$
  if $C_{\text{upd}} = \{v\}$ **and** $C_{\text{old}} = \{v\}$ **then**
    $A \leftarrow A \cup \{v\}$
    $c^i_{\text{upd}}(v) \leftarrow c^i_{\text{old}}(v)$
    continue
  else
    $q_{\text{new}} \leftarrow q_{\text{new}} \cup N_{\text{upd}}[v]$
  end if
end for
$q_{\text{new}} \leftarrow q_{\text{new}} \setminus A$
return $q_{\text{new}}, c^i_{\text{upd}}$

5.3.2 Correctness

Theorem 5.3 (Correctness of CAS). *Consider Algorithm 1 (*COLOR_UPDATE_PROPAGATION*) where* UPDATE_QUEUE *is replaced by* UPDATE_QUEUE_REMOVE_ASYMMETRIC_VERTICES*. Assume that in iteration i a processed vertex $v \in q$ satisfies*

$$C^i_{\text{old}}(v) = C^i_{\text{upd}}(v) = \{v\}.$$

Then omitting $N_{\text{upd}}[v]$ from the next queue and restoring $c^i_{\text{upd}}(v)$ to $c^i_{\text{old}}(v)$ does not affect the resulting partition: after iteration i, the coloring still induces the same partition as standard color refinement on G_{upd}.

Consequently, the algorithm produces a coloring equivalent to the stable coloring of G_{upd}, and thus correctly solves the dynamic color refinement problem (Definition 10).

Proof. Fix an iteration i and let $v \in q$ be a vertex that is asymmetric in both the old and updated partitions at iteration i. By Corollary 5.2.1, assigning v its original color $c^i_{\text{old}}(v)$ does not change any neighbor's color tuple in iteration $i+1$ compared to the from-scratch CR run, since the color of v is unique in both runs.

Therefore, v cannot influence the color refinement of its neighbors, and adding $N_{\text{upd}}[v]$ to the next queue is unnecessary. Skipping this update does not change any color tuple outside the queue.

All other vertices are processed exactly as in Algorithm 1. Thus, by Theorem 5.1, the resulting coloring in each iteration coincides with the corresponding iteration of the from-scratch refinement.

Hence, the pruning step preserves correctness in every iteration, and the final coloring is equivalent to the stable coloring of G_{upd}. By Definition 10, this implies that the algorithm correctly solves the dynamic color refinement problem. □

5.3.3 Complexity

For each vertex $v \in q_i$, the procedure performs only constant-time operations: two dictionary lookups and a test of whether $C^i_{\text{old}}(v)$ and $C^i_{\text{upd}}(v)$ are singletons. If v is not asymmetric, the algorithm adds v together with its neighbors to the new queue, which costs $O(\deg(v))$.

Hence, the cost of processing iteration i is

$$O\Big(|q_i| + \sum_{v \in q_i} \deg(v)\Big) = O\left(\sum_{v \in q_i} \deg(v)\right).$$

Thus, asymmetric-vertex pruning has the same asymptotic worst-case complexity as the basic queue update in CUP.

However, the pruning step can significantly reduce the size of q_i. An asymmetric vertex does not contribute to further propagation, and its neighbors are guaranteed not to change due to it. Consequently, CAS never increases the queue compared to CUP, and it may shrink it substantially. This reduces the number of vertices processed in subsequent iterations and may lead to a noticeable improvement in running time in practice.

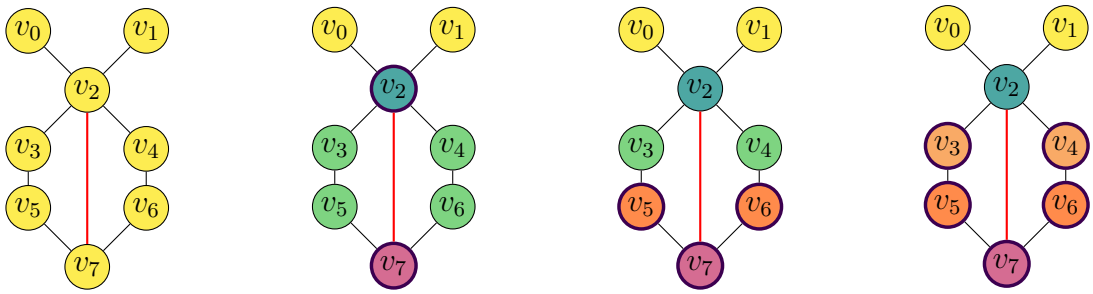
Motivation: positive examples

Asymmetric vertex separates the affected region from the rest of the graph. Consider a connected graph consisting of two large parts joined by a single asymmetric vertex - for instance, one with uniquely highest degree. If a modification occurs only in one part, and this connecting vertex remains unique, then the update cannot propagate across it, and the entire other part of the graph remains unaffected. The similar effect we get in a rooted tree.

Moderately dense random graphs. In many random graphs that are not extremely dense, the early refinement rounds typically produce a substantial number of asymmetric vertices. These vertices can act as barriers to propagation. As will be discussed later, such instances provide significant practical improvement for CAS compared to CUP.

Example

We now illustrate how CAS prunes asymmetric vertices in practice. The example below modifies the same graph as before by adding a single edge $\{v_2, v_7\}$. This update affects only one side of the graph, and vertex v_2 - which is already asymmetric in the original refinement - continues to act as a separator that prevents the change from propagating upward. The figure shows how the queue evolves over the iterations.



(a) Initial coloring, $q = \{v_2, v_7\}$

(b) Iteration 1. $q = \{v_5, v_6, v_7\}$

(c) Iteration 2. $q = \{v_3, ..., v_7\}$

(d) Iteration 3.

$$v = \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \end{bmatrix} \Bigg| \quad c^0_{upd} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \; c^1_{upd} = \begin{bmatrix} 1 \\ 1 \\ \cancel{2}\ \cancel{4}\ 2 \\ 3 \\ 3 \\ 3 \\ 3 \\ \cancel{3}\ 5 \end{bmatrix}, \; c^2_{upd} = \begin{bmatrix} 1 \\ 1 \\ 2 \\ 3 \\ 3 \\ \cancel{4}\ 5 \\ \cancel{4}\ 5 \\ \cancel{4}\ 6 \end{bmatrix}, \; c^3_{upd} = \begin{bmatrix} 1 \\ 1 \\ 2 \\ \cancel{3}\ 6 \\ \cancel{3}\ 6 \\ \cancel{4}\ 7 \\ \cancel{4}\ 7 \\ \cancel{5}\ 8 \end{bmatrix}$$

(e) Updated colorings for iterations 0-3. A single strike marks a color that changed in that iteration; the number next to it is the new color. A double strike indicates a color that was updated, identified as asymmetric, and then reverted.

Figure 5.2: Example illustrating the CAS pruning rule after adding the edge (v_2, v_7) (shown in red). Vertices with thick borders indicate the queue processed in each iteration. Vertex v_2 remains asymmetric in both the original and updated refinement and is therefore in iteration 1 removed from the queue.

In the original graph, vertex v_2 is already asymmetric in iteration 1, since it has the unique degree 4 among all vertices. After adding the edge $\{v_2, v_7\}$, its degree is still unique, and therefore v_2 is remains asymmetric.

Hence, the CAS rule applies: its recoloring is reverted to the old color, and its neighbors are not inserted into the next queue.

In contrast, in this vertex v_7 becomes asymmetric in the updated graph (again due to its unique degree), but it was not asymmetric in the original graph. Therefore v_7 and its neighbors must be inserted into the queue. After completing iteration 1, we obtain $q = \{v_5, v_6, v_7\}$.

In remaining iterations, the vertices v_0 and v_1 are succesfully separated from the propagation.

5.4 COR: pruning unchanged orbits

COR algorithm extends the idea behind CAS from individual asymmetric vertices to entire color classes. During an update, it may happen that all vertices of an old color class are temporarily recolored but end the iteration in exactly the same way. When this occurs, the class has no influence on the rest of the graph, since all its vertices present identical updated tuples to their neighbors.

A simple example illustrates this observation. Consider a graph consisting of a large connected

component and two isolated vertices. Initially, the isolated vertices form one color class (degree 0). If we add an edge between them, their degrees increase in the same way, and refinement groups them together again. Although this class is not asymmetric, it does not affect any other vertex and does not cause further propagation.

This leads to the central rule of COR: if, in some iteration, an entire color class changes uniformly compared to the original refinement and all its vertices are contained in the current queue, then the class can be removed from the queue and its original color restored. Only classes whose colors actually change continue to propagate updates.

5.4.1 Pseudocode

We now present the algorithm COR, another alternative to UPDATE_QUEUE in Algorithm 1.

Algorithm 6 COR (UPDATE_QUEUE_REMOVE_UNCHANGED_ORBITS)

Input: $G_{\text{upd}},\ q,\ \gamma^i_{\text{upd}},\ D^i_{\text{upd}},\ c^i_{\text{upd}},\ \text{old_iterations}[i]$
Output: Updated queue q_{new}, updated coloring c^i_{upd}

$U \leftarrow \emptyset$ ▷ set of unchanged color classes (orbits)
$A \leftarrow \emptyset$ ▷ set of asymmetric vertices to exclude
$\text{seen_tuples} \leftarrow \emptyset$
for each $v \in q$ **do**
  if $\gamma^i_{\text{old}}(v) \in \text{seen_tuples}$ **or** $\gamma^i_{\text{upd}}(v) \in \text{seen_tuples}$ **then**
    continue
  end if
  $C_{\text{old}} \leftarrow D^i_{\text{old}}[\gamma^i_{\text{old}}(v)]$
  $C_{\text{upd}} \leftarrow D^i_{\text{upd}}[\gamma^i_{\text{upd}}(v)]$
  $\text{seen_tuples} \leftarrow \text{seen_tuples} \cup \{\gamma^i_{\text{old}}(v), \gamma^i_{\text{upd}}(v)\}$
  if $C_{\text{upd}} \subseteq q$ **and** $C_{\text{upd}} = C_{\text{old}}$ **then** ▷ orbit unchanged and entirely inside q
    $U \leftarrow U \cup C_{\text{upd}}$ ▷ mark whole orbit for removal
    for each $w \in C_{\text{upd}}$ **do**
      $c^i_{\text{upd}}(w) \leftarrow c^i_{\text{old}}(w)$
    end for
    if $|C_{\text{upd}}| = 1$ **then**
      $A \leftarrow A \cup \{v\}$ ▷ store asymmetric vertex
    end if
  else
    $q_{\text{new}} \leftarrow q_{\text{new}} \cup N_{\text{upd}}[v]$
  end if
end for
$q_{\text{new}} \leftarrow q_{\text{new}} \setminus A$
return $q_{\text{new}},\ c^i_{\text{upd}}$

Remark. For asymmetric vertices the pruning is permanent: if a vertex v is asymmetric at iteration i, then it remains asymmetric in all later iterations. Thus, when CAS removes v from the queue, we not only avoid adding $N_{\text{upd}}[v]$ to the new queue, but v we can also remove v from the queue in general.

For COR this is fundamentally different. Suppose we a color class C is identified as unchanged in iteration i. When constructing q_{i+1}, we can ommit adding to it $N_{\text{upd}}[C]$. However, this does *not* mean that the vertices of C are excluded from the queue entirely. If a vertex $u \in C$ lies in

the neighborhood of another vertex $w \in q_i$ whose tuple genuinely changed, then u must still be added to q_{i+1}, even if the entire class C was removed as a unit.

Hence COR removes a class only from the *current* propagation step but does not prevent its vertices from being reintroduced when required by other affected vertices. To strengthen pruning, COR therefore also tracks asymmetric vertices separately, since those provide permanent barriers to propagation.

5.4.2 Correctness

Theorem 5.4 (Correctness of COR pruning)**.** *Let C be a color class in iteration i of the updated refinement such that:*

1. *$C = C^i_{\text{upd}}(v) = C^i_{\text{old}}(v)$ for all $v \in C$, i.e. the class is unchanged, and*
2. *$C \subseteq q$, i.e. all vertices of the class lie inside the current queue.*

Then restoring the old colors of all vertices in C and removing C from the queue produces exactly the same coloring in iteration $i+1$ as the standard color refinement on G_{upd}.

Consequently, the algorithm produces a coloring equivalent to the stable coloring of G_{upd}, and thus correctly solves the dynamic color refinement problem (Definition 10).

Proof. Let χ^i_{cr} denote the coloring obtained by running standard color refinement on G_{upd}, and let χ^i_{upd} be the coloring produced by CUP on G_{upd} *before* applying COR pruning in iteration i. By the correctness of CUP (Theorem 5.1), χ^i_{upd} and χ^i_{cr} induce the same partition.

Now let C satisfy conditions (1) and (2). By (1), C is the same class in the original and updated refinements, and all its vertices already have the correct color. Restoring the old colors on C therefore does not change the partition, and all neighbors of C see the same multisets as in the CUP run. Thus the color tuples in iteration $i+1$ are unchanged, and χ^{i+1}_{upd} induces the same partition as χ^{i+1}_{cr}.

It remains to show that removing C from the queue does not affect the result of the refinement. If C were not pruned, the algorithm would enqueue $N_{\text{upd}}[C]$ and recompute their tuples in iteration $i+1$. However, their multisets of neighbor colors already match those in the CUP run, so this recomputation would produce the same values. Pruning C therefore only avoids redundant work and does not change the resulting partition.

Since this argument applies in every iteration where COR pruning is used, the overall refinement produced by CUP with UPDATE_QUEUE_REMOVE_UNCHANGED_ORBITS induces the same partition as standard color refinement on G_{upd}. By Definition 10, this implies that the algorithm correctly solves the dynamic color refinement problem. □

5.4.3 Complexity

In iteration i, the COR queue update processes each vertex $v \in q_i$ once. For every such v, it performs a constant number of dictionary lookups (to obtain $C^i_{\text{old}}(v)$ and $C^i_{\text{upd}}(v)$), a membership test in $\mathrm{seen_tuples}$, and constant-time set updates, all in $O(1)$ time per vertex. Grouping the vertices of q_i by color class via the dictionaries D^i_{old} and D^i_{upd} therefore costs $O(|q_i|)$ in total.

For each color class that intersects the queue, the algorithm considers only the subset contained in q_i, compares it with the corresponding subset from the old iteration, and, if the class is unchanged, restores the old colors of its vertices. Although a color class may extend outside q_i, the subsets examined by the algorithm are disjoint and together cover all of q_i, so these comparisons and possible color restorations also take $O(|q_i|)$ time.

The construction of the next queue q_{i+1} consists of adding the neighbors of vertices in q_i that were not pruned, which costs

$$O\Big(\sum_{v \in q_i \setminus U} \deg(v)\Big) \subseteq O\Big(\sum_{v \in q_i} \deg(v)\Big).$$

Thus, the total running time of COR in iteration i is

$$O\big(|q_i| + \sum_{v \in q_i} \deg(v)\big) = O\left(\sum_{v \in q_i} \deg(v)\right),$$

the same asymptotic order as in CUP and CAS.

The cost of tuple computation remains unchanged, so the overall complexity of COR is obtained from that of CUP by replacing the queue sizes q_i by their pruned versions.

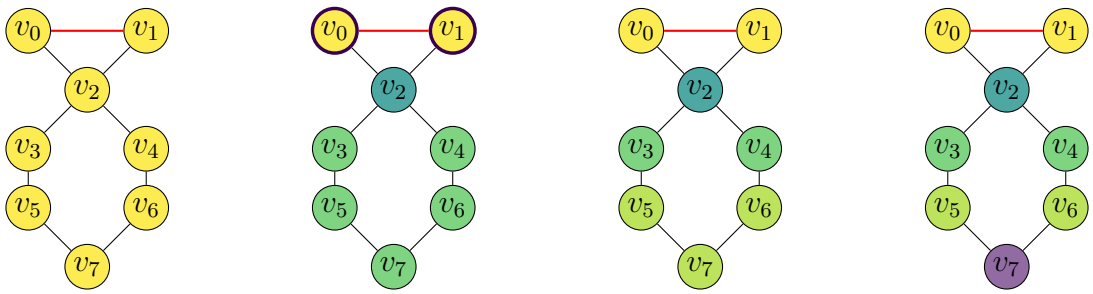
In practice, however, COR can substantially reduce the sizes of the queues q_i through class-based pruning and therefore often achieves lower running times on real graphs.

Examples

Unchanged color class

We begin with an example illustrating the basic situation in which an entire color class changes uniformly and therefore does not propagate any updates. This is the best case for CUP: the modification does not affect the partition at all, CUP detects this, and the entire refinement history can be reused from the stored iterations.

In the graph below, the newly added edge $\{v_0, v_1\}$ affects only its endpoints, both of which remain in the same color class throughout the refinement. Since this class is unchanged and fully contained in the initial queue, COR removes it immediately, and the queue becomes empty. No further vertices are affected, and the refinement proceeds exactly as in the original graph.



(a) Initial coloring $q = \{v_0, v_1\}$ **(b)** Iteration 1 $q = \{\}$ **(c)** Iteration 2 $q = \{\}$ **(d)** Iteration 3, Iteration 4 $q = \{\}$

Figure 5.3: Updating CR colorings using COR algorithm. After the first iteration the queue becomes empty and hence no further changes are needed.

Stable class with propagation

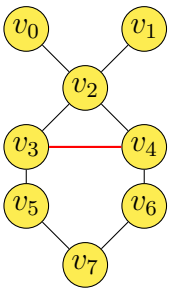
We now consider an example in which a color class becomes stable early, but its effect on the graph is not obvious.

After inserting the edge $\{v_3, v_4\}$, the vertices v_3 and v_4 become symmetric and already form a two-element color class in iteration 1. However, this update breaks the old color class $\{v_3, \ldots, v_7\}$,

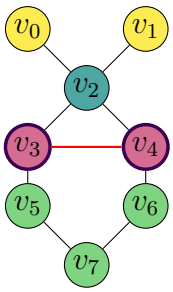
so its entire neighborhood must enter the queue.

By the end of iteration 2, the class $\{v_3, v_4\}$ has stabilized and could, in principle, be pruned. Nevertheless, their neighbors v_5 and v_6 still change color in this iteration Because these changes must be propagated one layer further, the vertices v_3 and v_4 must remain in the queue. Although their own colors do not change, v_5 and v_6 changed their color class and they can potentially by influenced by that. In contrast, v_2 becomes asymmetric and is therefore removed from the queue.

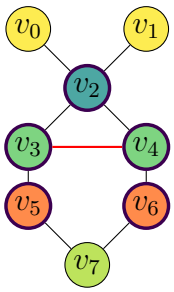
Only in iteration 3 (which is the last iteration for this graph) does the updated partition match the original one, and the final queue becomes empty.



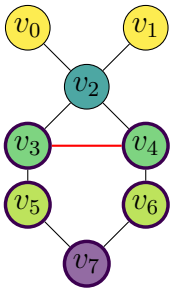
(a) Initial coloring. $q = \{v_3, v_4\}$



(b) Iteration 1. $q = \{v_2, ..., v_6\}$



(c) Iteration 2 $q = \{v_3, ...v_6\}$



(d) Iteration 3 $q = \{\ \}$

$$v = \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \end{bmatrix} \quad c^0 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \; c^1 = \begin{bmatrix} 1 \\ 1 \\ 2 \\ 3\ 4 \\ 3\ 4 \\ 3 \\ 3 \\ 3 \end{bmatrix}, \; c^2 = \begin{bmatrix} 1 \\ 1 \\ 2\ 5\ 2 \\ 3\ 6\ 3 \\ 3\ 6\ 3 \\ 4\ 7 \\ 4\ 7 \\ 4 \end{bmatrix}, \; c^3 = \begin{bmatrix} 1 \\ 1 \\ 2 \\ 3\ 5\ 3 \\ 3\ 5\ 3 \\ 4\ 6\ 4 \\ 4\ 6\ 4 \\ 5\ 7\ 5 \end{bmatrix}$$

(e) The updated colorings of respective iterations.

Figure 5.4: Update CR colorings using COR algorithm.

This example illustrates the main difference between CAS and COR: CAS operates on individual asymmetric vertices, while COR considers the entire color classes. Even if a class remains unchanged, it cannot be removed too early if it lies in the neighborhood of vertices whose updates still propagate to the next round of refinement.

5.5 Comparison of CUP, CAS, and COR

The three update strategies share the same overall goal: reuse the original refinement as much as possible and restrict recomputation to the part of the graph actually affected by the edge modification. They differ only in how selectively they prune vertices from the propagation process.

CUP follows the simplest update strategy: it treats every vertex that may be influenced by the modification as potentially relevant. It recomputes color tuples only for the vertices in the current queue and propagates changes by repeatedly adding their neighbors, without performing any pruning. As a result, every vertex that could be affected remains in the queue until the refine-

ment stabilizes. Despite its conservative nature, CUP can still perform well in situations where the update is contained entirely within a disconnected component, since the propagation remains restricted to that component.

CAS refines CUP by identifying asymmetric vertices. Whenever a vertex is unique in both the original and updated refinements at some iteration, it cannot influence its neighbors in later rounds. CAS removes such vertices from the queue and does not propagate through them further. This never increases the work compared to CUP and can substantially reduce queue sizes in graphs where color refinement quickly produces many asymmetric vertices.

COR extends the pruning idea from individual vertices to entire color classes. If a whole class behaves exactly as in the original refinement and all of its vertices are currently in the queue, COR restores their old colors and removes the entire class from the queue. This can eliminate large groups of vertices at once, particularly in graphs with substantial symmetry or in early iterations of color refinement.

However, such classes may still be affected later in the process: even if a class is unchanged in iteration i, it may split in iteration $i+1$ if one of its neighbors changes. Thus a class removed from the queue can reappear if it lies in the neighborhood of a vertex whose color changes. To avoid missing vertices whose colors are permanently fixed, COR additionally uses the CAS rule of excluding asymmetric vertices from the queue.

Complexity and practical behavior. All three methods have the same asymptotic worst-case complexity:

$$O\Big(\sum_i \sum_{v \in q_i} \deg(v) \log \deg(v)\Big) \quad \text{or} \quad O\Big(\sum_i \sum_{v \in q_i} \deg(v)\Big),$$

depending on the implementation of color tuple computation. Here q_i denotes the queue in iteration i of the update.

CUP performs no pruning and thus gives an upper bound on the amount of work. CAS can only decrease the size of the queues by removing asymmetric vertices, and COR can reduce them even further by discarding entire color classes. In practice, the benefit of CAS and COR depends on structural properties of the graph, such as density and diameter, but most importantly on the type and scale of the update.

6 Application

This section studies how the three update algorithms - CUP, CAS, and COR - perform in practice. We apply them to datasets of randomly generated graphs, modify each graph by a small change, and measure how much work the algorithms perform when updating the existing color refinement.

The goal is to see how the theoretical pruning rules compare with practical savings and how the structure of a graph influences the amount of recomputation needed.

6.1 Experiment Setting

Implementation

All experiments were performed in Python using the `networkx` library. The update step of color refinement was implemented using the hash-based encoding of color tuples described in Appendix, following the approach of [22]. In particular, color tuples were computed with matrix-vector multiplication and encoded using the prime/log scheme.

This optimization affects only the internal representation of color tuples and does not change the refinement rule or the resulting vertex partition.

The full code base - including dataset generation, computation of the original CR colorings, execution of all update algorithms, and plotting scripts - as well as the generated datasets are available in supplementary materials.

Datasets

The experiments were conducted on randomly generated graphs, divided into two groups:

1. random trees;
2. Erdős-Rényi random graphs $G(n, p)$ for several values of p.

For each group, 250 graphs were generated. The number of vertices n was drawn uniformly at random from the interval $100 \leq n \leq 500$.

Random trees. For each sampled value of n, a uniformly random labeled tree on n vertices was generated using `nx.random_labeled_tree`.

Erdős-Rényi graphs. Random graphs were generated using `nx.fast_gnp_random_graph`. For each value $p \in \{0.005, 0.01, 0.05\}$, 250 graphs were produced. As in the tree case, n was drawn uniformly from $[100, 500]$ before generating the graph.

Procedure

For every graph in the datasets, the static CR algorithm *(SCR)* was run once to compute the full refinement history and store the corresponding *old_iterations* object. Each graph was then ran-

domly modified (described in detail in the further sections). Finally, all three update algorithms were executed on the modified graph. During each run, preformance statistics was gathered.

Measuring Results

Since absolute running time depends strongly on practical choices, we evaluate the algorithms using implementation-independent quantities. Let $q_0, q_1, \ldots, q_k$ denote the queues generated during the execution of CUP, CAS, or COR while using the old iterations ($k = min(T_{old}; T_{upd})$). We measure performance using the following measures:

1. **Total work**
$$W = \sum_{i=0}^{k} |q_i|.$$
This counts the total number of vertices processed across all iterations.
2. **Queue length** For each dataset we record the sizes of the queues $|q_i|$ throughout the update process. For trees we report their average values; for random graphs we normalize queue sizes by n and compare them across iterations.
3. $\boldsymbol{\Delta C}$: **number of recolored vertices** This denotes the number of vertices whose final color differs from their original color under standard color refinement. Note that this quantity may vary between update algorithms, depending on their effectiveness in avoiding unnecessary recoloring of unchanged vertices.
4. $\boldsymbol{\Delta T}$: **change in number of iterations** Difference between the number of iterations of SCR and update algorithm run on updated graphs. Note: in this results, in T the last ("check") iteration is included.

6.2 Random Trees: removing an edge

As a first test case we examine updates on random trees. We look closer at a simple modification of a tree, which is removing from it one edge. Even though the change is small compared to the number of edges in the tree, it can be crutial to the structure of a tree. If the deleted edge is not a leaf, it divides the tree into two disconnected commponents. At the same time, trees often contain long symmetric branches and therefore require several refinement rounds, which makes them a meaningful setting for comparing the update behavior of CUP, CAS, and COR.

For this experiment, 250 random labeled trees with $100 \leq n \leq 500$ were generated using `nx.random_labeled_tree`. From each tree one edge was selected uniformly at random and removed; the two endpoints of this edge formed the initial set S for the update.

For every such update we recorded the quantities introduced in the previous section: total processed vertices, queue sizes over iterations, number of recolored vertices, and the number of refinement rounds after the update

6.2.1 Total proceeded vertices

The total number of processed vertices remains small for all three algorithms. Compared to the standard color refinement (SCR), which processes more than 10^3 vertices, all proposed methods achieve a reduction of over an order of magnitude. CUP processes on the order of slightly more than 10^2 vertices per update, with a mild increasing trend as n grows. CAS and COR perform substantially less work, typically staying below 30 processed vertices in total.

This already indicates that the effect of removing a single edge from a tree is highly localized. Even CUP, without doing any pruning, does not expand far into the graph, most likely because the diameter of the affected region is small relative to the full tree. CAS and COR perform well in this situation: by pruning symmetric vertices and unchanged color classes, they reduce the update workload by nearly an order of magnitude.

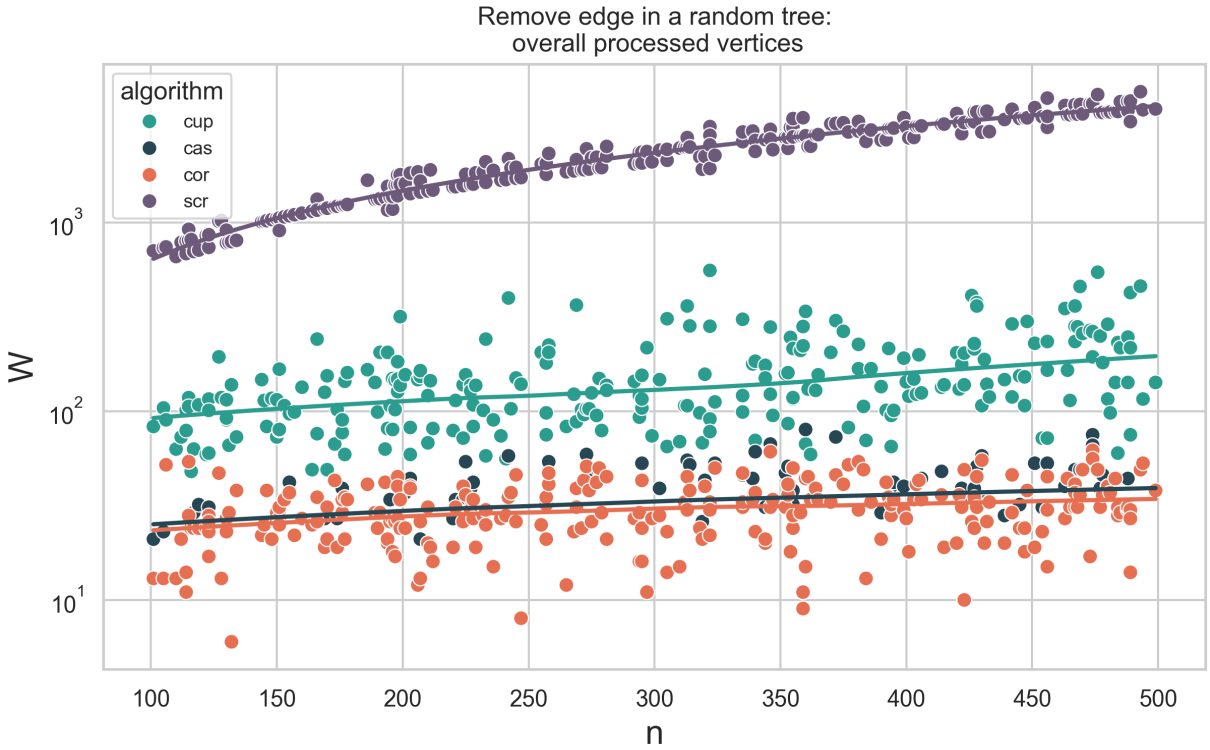


Figure 6.1: Total processed vertices W for CUP, CAS, COR, and SCR on the trees dataset. For standard color refinement, we assume that in each iteration n vertices are processed. The y-axis is shown on a logarithmic scale. A LOWESS curve is fitted to each set of results.

Overall, all three algorithms behave very well on this update operation, with CAS and COR offering a particularly strong reduction of processed vertices.

6.2.2 Average queue length

The structure of a tree gives good condition for building queues with propagation. For CUP, the average queue size exhibits a slow linear growth, but remains significantly below n vertices proceeded by SCR. Importantly, this quantity shows no systematic growth with n: even in trees with 500 vertices, the update rarely propagates far from the modified edge.

CAS and COR keep the queue even smaller. Across all tested sizes, the average queue length for both algorithms stays below 10 vertices, with COR being slightly smaller than CAS.

Overall, all three algorithms maintain short queues, but CAS and COR restrict the affected region far more effectively, leading to significantly less work per iteration.

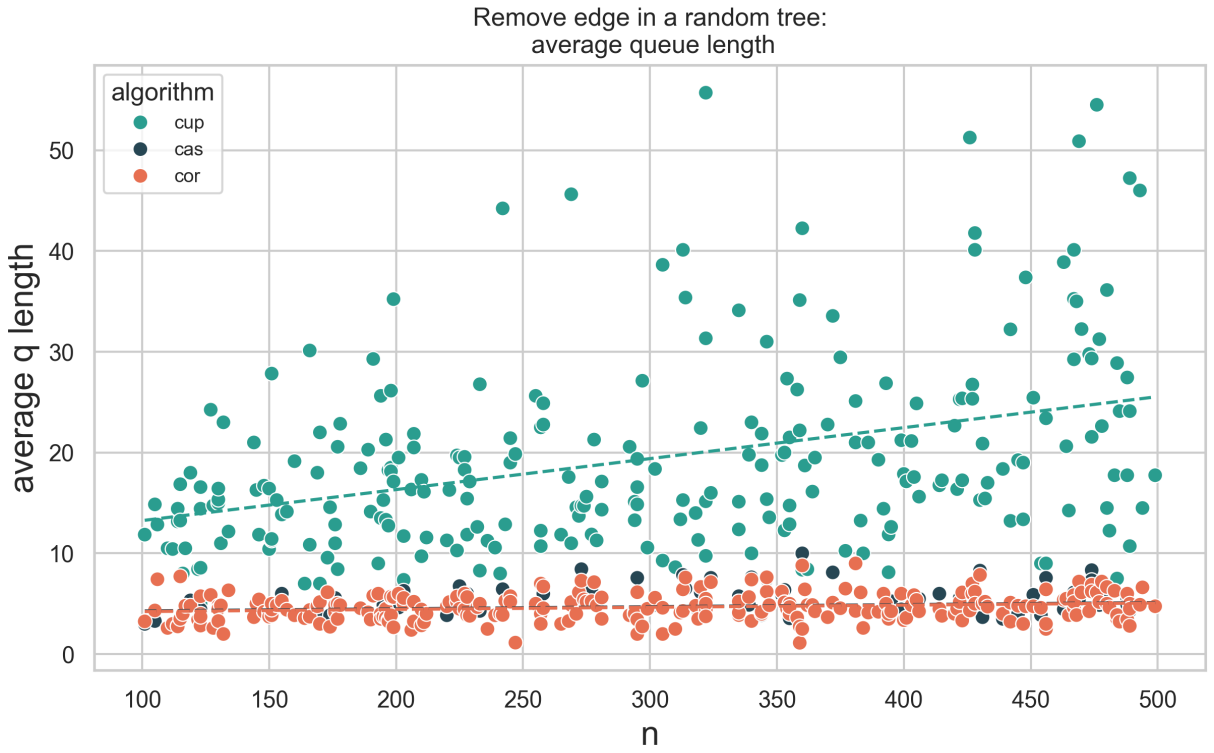


Figure 6.2: Average length of the queue for CUP, CAS, and COR on trees dataset. A LOWESS curve is fitted to each set of results.

6.2.3 Changed vertices in the final coloring

To evaluate how far a local modification propagates, we measure the number of vertices whose color changes in the final stable coloring, denoted ΔC. Since removing a single edge in a tree is a highly local operation, one expects the effect on the coloring to remain limited.

The results are summarized in Fig. 6.3. The left panel shows CUP, while the right panel compares CAS and COR.

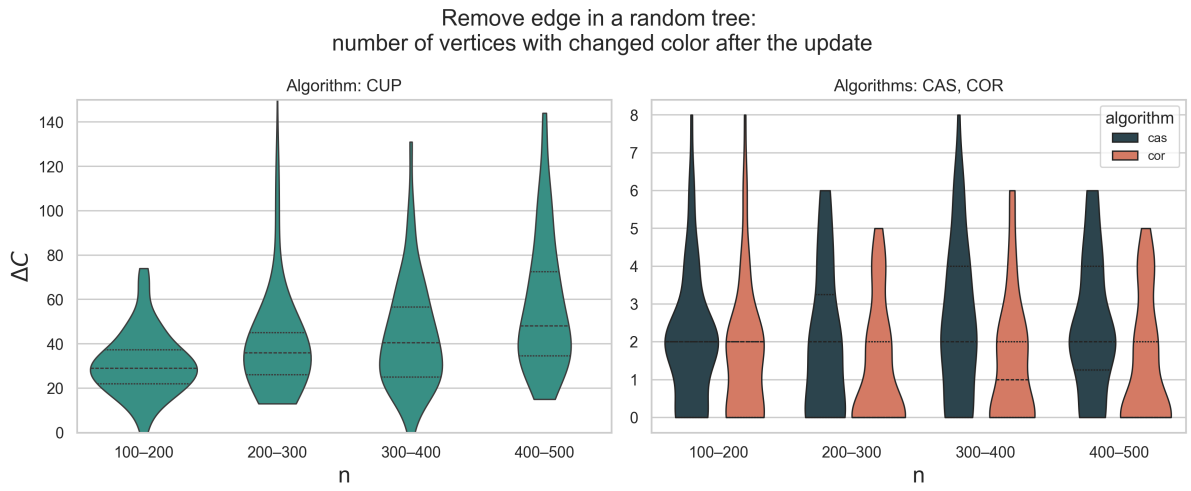


Figure 6.3: Number of vertices with changed color (ΔC) after removing one edge, for CUP, CAS, and COR on the tree dataset. The dotted lines on the violin plots mark the 25th, 50th and 75th percentils of distributions. In the second plot, five outlier results are excluded from the plot.

For CUP, the number of recolored vertices varies widely and increases with the size of the tree, typically ranging from more than twenty to around sixty for medium-sized trees, with occasional larger updates. However, even for CUP the update does not reach the whole tree. Even for trees with n between 400 and 500, the median of changed vertices stays below 60.

In contrast, CAS and COR affect only a very small number of vertices. Across all tree sizes, the distributions are strongly concentrated between 0 and 2, and recolorings beyond a few vertices are extremely rare. COR consistently gives best results, producing the smallest updates.

To illustrate the scaling behavior, Fig. 6.4 plots the value of ΔC as a function of n. For all three algorithms, the trend remains essentially constant as n increases.

While CUP exhibits a larger variance, with a median of about 36 recolored vertices, CAS stays well below 10, with a median of 2. COR performs even better, with a median of only 1. This indicates that the relative effect of removing an edge becomes negligible as the size of the tree grows.

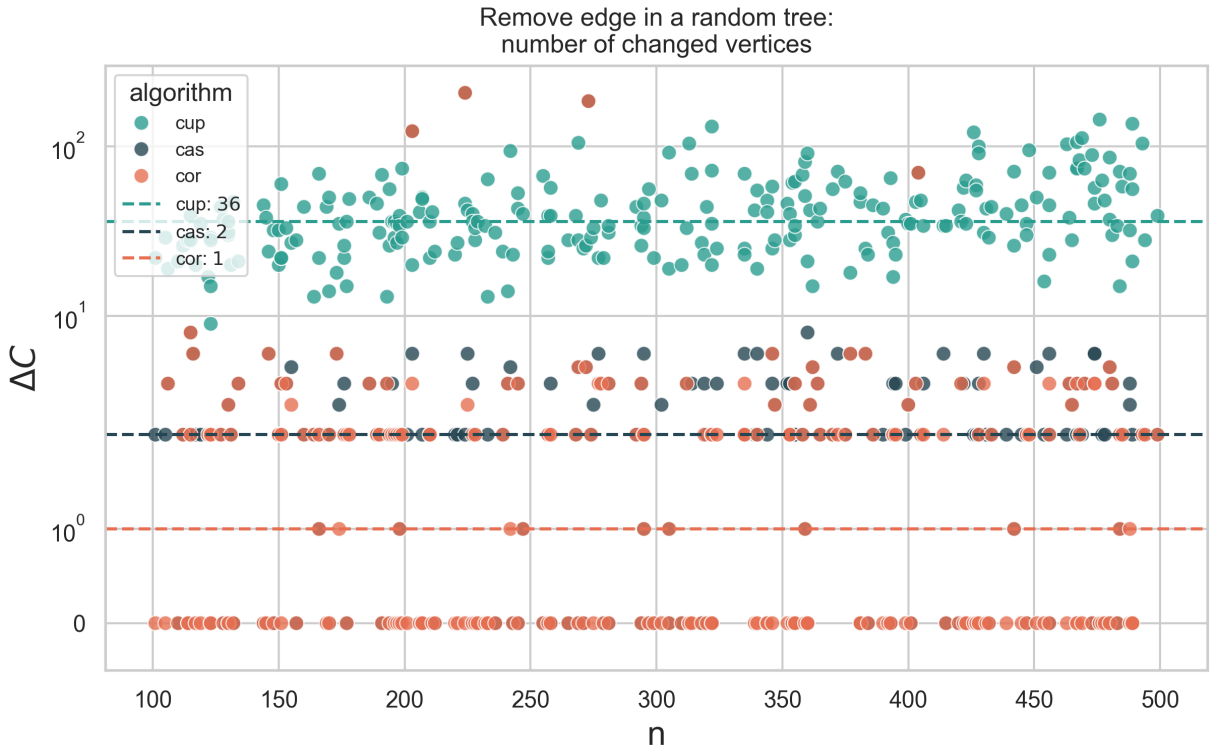


Figure 6.4: Number of changed vertices ΔC after removing one edge from a random tree. Dashed reference lines are medians of the datasets.

Finally, Table 6.1 summarizes these observations numerically. CUP always changes some colors and has a median of 36 recolorings, with large variability. CAS reduces the median to two and performs no recoloring in roughly 28% of the trees. COR gives even better results: in 43% of cases no vertex changes color, and 75% require at most only two changes.

algorithm	median	Q1–Q3	max	$p(\Delta C > 0)$
cup	36	[26.25, 50.75]	208	0.992
cas	2	[0, 4]	207	0.716
cor	1	[0, 2]	207	0.528

Table 6.1: Change in the number of recolored vertices ΔC after removing one edge from a random tree. Q1 and Q3 denote the 25th and 75th percentiles, respectively. It is worth noting that ΔC can be zero even for CUP; this occurs when the updated graph requires more iterations than the original one, but the resulting partition remains unchanged.

Overall, these results show that the effect of removing an edge in a tree is indeed highly localized. On the one hand, this is can be expected: the update operation stays the same applied to different tree sizes. On the other hand, deleting an edge can significantly change the structure of a tree by splitting it into two connected components, so in principle the refinement could change more broadly. CUP propagates this change noticeably, but still far from globally. Pruning in CAS, and especially COR, reduces unnecessary propagation, making recoloring almost negligible in most cases.

6.2.4 Number of iterations

To interpret how the update affects the number of iterations, it is useful to recall how many iterations standard color refinement requires on the original trees. Typical values are shown in Table 6.2; for most trees, the process stabilizes after about eight rounds.

Table 6.2 summarizes the change in iteration count $T_\Delta = T_{\text{up}} - T_{\text{scr}}$ after deleting one edge. For CUP, T_Δ is always zero, i.e., the update does not change the refinement depth.

For CAS and COR, the behavior differs. In both cases, the number of iterations typically remains unchanged, but increases do occur with non-negligible probability. For CAS, the iteration count increases in about 26% of the runs, while for COR this happens in roughly 45% of the cases. When a change occurs, it is usually small, with an increase of at most a few iterations. However, there are cases when this number can be reduced significantly, reaching six iterations.

Overall, the update rarely affects the refinement depth substantially, and even when it does, the change remains limited.

algorithm	median	Q1–Q3	max	$p(\Delta T \neq 0)$
scr	8	[7, 8]	10	–
cup	0	[0, 0]	0	0.000
cas	0	[0, 1]	6	0.260
cor	0	[0, 2]	6	0.452

Table 6.2: Change in the number of iterations ΔT after removing one edge from a random tree. The first row shows the typical number of refinement iterations T_{scr} for standard color refinement.

Summary

Removing a single edge from a tree is a very specific update, and the results should be interpreted with this in mind. The original number of CR iterations is nearly constant across all tested tree sizes, and this already limits how far any update can propagate. In principle, deleting an edge could substantially change the structure of a tree by splitting it into two components, so a wider effect is still possible.

In practice, however, the effect remains highly local: only a small fraction of vertices are ever recolored. Additionally, situations in which the final coloring does not change after the update can often be detected early, allowing the update procedure to terminate sooner and reducing the required number of refinement rounds.

This experiment therefore supports the conclusion that updating the coloring, instead of recomputing it from scratch, can be highly effective - even in cases where the global structure of the graph changes.

6.3 G(n,p) Random Graphs

The second part of the evaluation considers Erdős–Rényi graphs $G(n, p)$.

For each $p \in \{0.005, 0.01, 0.05\}$, I generated 250 graphs with n chosen uniformly from $[100, 500]$ using the NetworkX generator `nx.gnp_random_graph`, resulting in a dataset of 750 graphs in total. Each graph was then perturbed by modifying a small fraction of its edges. The scale of the modification is defined by a parameter $\alpha \in \{0.001, 0.005, 0.01\}$: given $m = |E|$, I choose

$k = \max\{1, \lfloor \alpha m \rfloor\}$ edges to change, delete $\lfloor k/2 \rfloor$ of them, and insert the remaining ones at random. The resulting perturbed graph is denoted G^{α}_{up}.

6.3.1 Total processed vertices

The total number of processed vertices increases with the density of the graph. For small values of p, both CAS and COR process only a small fraction of the vertices; their values remain well below 0.5, and for the smallest densities even show a slight decrease with n. In contrast, CUP exhibits significant variance in this regime, although the overall tendency still increases with n.

For larger edge densities the behavior becomes more regular. CUP follows an almost linear trend in n, closely tracking a curve that remains below $2n$ and gradually flattens as it approaches this bound. CAS and COR show a similar qualitative behavior, but at a much lower level: their curves tend towards values slightly above n, rather than approaching $2n$.

This behavior can be explained by the iteration structure of color refinement. Although CAS and COR may initially reach a large portion of the graph, most of these vertices are eliminated in the following iteration as they become asymmetric. Consequently, W/n remains close to 1 for both algorithms.

Finally, all three update methods significantly outperform standard color refinement (SCR), shown in Fig. 6.6, where in each iteration the entire graph is processed.

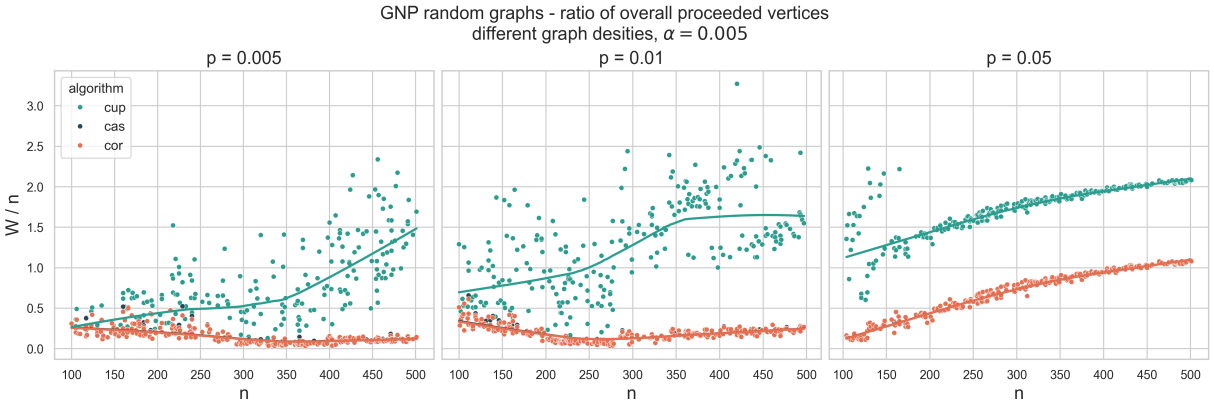


Figure 6.5: Total processed vertices W for CUP, CAS, and COR on $G(n,p)$ graphs ($p \in \{0.005, 0.01, 0.05\}$) with perturbation level $\alpha = 0.005$, normalized by n. Each dataset is fitted with LOWESS curve.

On the other hand, let us examine how the scale of the update influences the total number of processed vertices. In Fig. 6.7 we compare graphs with $p = 0.01$ for different values of α.

For very small perturbations ($\alpha = 0.001$), the behavior differs significantly between the algorithms. CUP exhibits a large variance and a non-monotone trend: the ratio W/n increases up to about $n \approx 300$, where the fitted LOWESS curve reaches values close to 1.5, and then decreases again for larger n. In contrast, CAS and COR remain very stable and small; their values stay well below 0.5 and even show a slight decrease with n.

For $\alpha = 0.005$ and $\alpha = 0.01$, the behavior becomes more regular and similar across both settings. CUP shows a clear growth in W/n, with the increase slowing down around $n \approx 350$. The variance decreases with α, although it remains noticeable. CAS and COR still process only a small fraction of the graph: their values remain close to zero, typically below 0.5, with only a slight increasing trend as n grows.

Overall, even for larger perturbations, CAS and COR consistently limit the spread of the update, while CUP exhibits a broader and more variable propagation pattern.

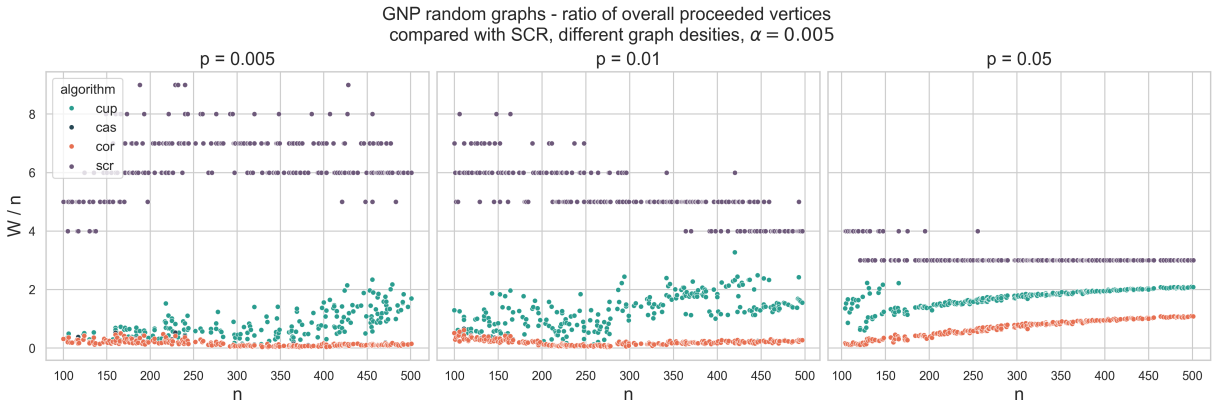


Figure 6.6: Normalized number of total processed vertices W/n for CUP, CAS, and COR on $G(n, p)$ graphs for different p with perturbation level $\alpha = 0.005$, compared with the standard color refinement. We assume, that in SCR in each iteration the whole graph is proceeded, so $W/n = T_{scr} * 1$.

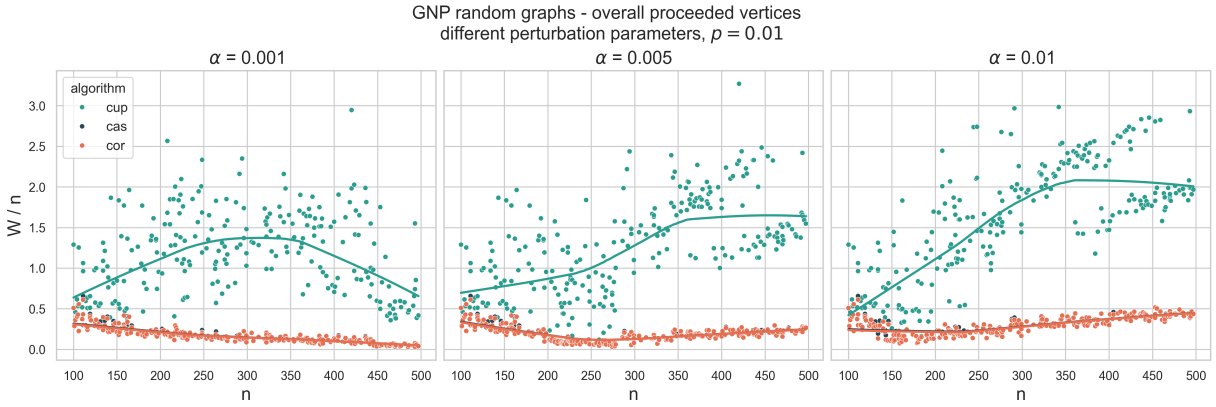


Figure 6.7: Total processed vertices W for CUP, CAS, and COR on $G(n, 0.01)$ graphs for perturbation levels $\alpha \in \{0.001, 0.005, 0.01\}$, normalized by n.

It is worth noticing, that across all values of p and α, CAS and COR behave almost identically in terms of total work. This indicates that most pruned vertices are asymmetric and therefore removed by both algorithms.

6.3.2 Number of iterations

To interpret the behavior of the queue lengths in the later subsection, it is useful to first examine the refinement depth of standard color refinement on the updated graphs. Table 6.3 shows the distribution of the number of refinement iterations for the three edge densities used in the experiments.

p	median	Q1–Q3	min–max	mean ± std
0.005	6	[6, 7]	4–9	6.39 ± 0.89
0.01	5	[5, 6]	4–8	5.29 ± 0.87
0.05	3	[3, 3]	3–4	3.09 ± 0.29

Table 6.3: Typical number of refinement iterations T_{scr} on random graphs for different p.

These values indicate three qualitatively different regimes. For very sparse graphs ($p = 0.005$),

the iteration depth still shows noticeable variability. For $p = 0.05$, almost all graphs stabilize after three iterations (two proper refinement rounds), which matches the behavior predicted in [3]. The intermediate density $p = 0.01$ is the most informative case: the refinement depth is neither trivial nor fully concentrated, and thus best reflects how the update affects the propagation.

For this reason, the analysis of the iteration behavior focuses on $p = 0.01$. Table 6.4 summarizes the values of $T_\Delta = T_{\text{upd}} - T_{\text{scr}}$ for the three perturbation scales α.

For CUP, T_Δ is always zero for all values of α, meaning that the update does not affect the refinement depth. In contrast, CAS and COR can change the number of required iterations. For $\alpha = 0.001$ and $\alpha = 0.005$, the median increase is one iteration, and a change occurs in more than 50% of the runs. For $\alpha = 0.01$, the median drops to zero, although changes still occur with non-negligible probability (around 40–45%). In all cases, the change in the number of iterations remains small in absolute terms, typically within one or two rounds, with maxima up to four.

algorithm	median	Q1–Q3	$p(T_\Delta \neq 0)$	max
$\alpha = 0.001$				
cup	0	[0,0]	0.000	0
cas	1	[0,1]	0.536	3
cor	1	[0,2]	0.556	4
$\alpha = 0.005$				
cup	0	[0,0]	0.000	0
cas	1	[0,2]	0.592	3
cor	1	[0,2]	0.604	3
$\alpha = 0.01$				
cup	0	[0,0]	0.000	0
cas	0	[0,1]	0.424	3
cor	0	[0,1]	0.472	4

Table 6.4: Change in the number of refinement iterations T_Δ for different perturbation levels α.

In summary, while CUP preserves the number of iterations exactly, CAS and COR may alter it in a substantial fraction of cases. Even a change by one iteration can be significant given the small overall depth (typically between three and six rounds), so pruning can meaningfully affect the refinement process, although the absolute difference always remains low.

6.3.3 Average queue lengths through the refinement process

To understand how the update propagates through different stages of the refinement, we analyse the evolution of the queue sizes during the iteration sequence. Since the total number of iterations varies across runs, a direct comparison of queue lengths by absolute iteration index is not very informative. Instead, each refinement run is divided into four equal relative phases (0-25%, 25-50%, 50-75%, 75-100%), and for each phase we compute the median queue size (normalised by n) over all iterations.

Figure 6.8 shows the results for graphs with $p = 0.01$ and three perturbation parameters $\alpha \in \{0.001, 0.005, 0.01\}$.

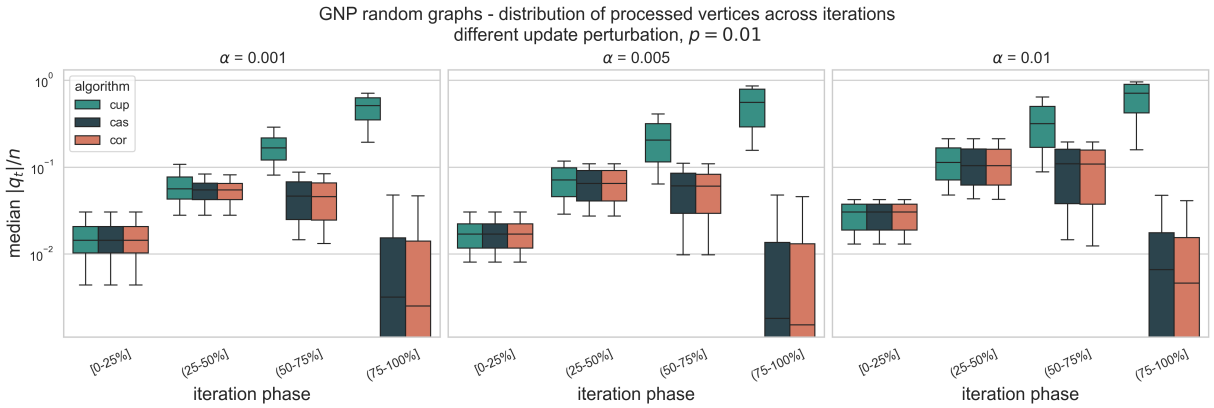


Figure 6.8: Median queue sizes (normalised by n) across four relative refinement phases for $G(n, 0.01)$ graphs under three perturbation levels. The boxplots summarise the distribution over all runs: the box spans the interquartile range, the line marks the median, and whiskers show the 10th and 90th percentiles. The y-axis is logarithmic.

Across all perturbation levels the qualitative behavior of the queue is the same. In the first phase the queue contains only the initial input set S, and then its closest neighborhood. In the middle phases the queue grows and reaches its maximum; this is the moment when many vertices become temporarily relevant before the refinement begins to stabilise. In the final phase the queue becomes small again.

This is expected for random graphs, where most vertices become asymmetric very quickly. Consequently, when the perturbation is small and the graph is not too dense, the total number of proceeded vertices of CAS and COR is largely determined by the height of this middle peak. Most of these vertices are pruned again in the subsequent phase.

For small perturbations the queues remain tiny relative to the size of the graph, typically well below $1/10$ of all vertices. The comparison between CUP and the pruned variants is particularly clear: CUP shows the full propagation of the change and its queues approach a fraction close to one in the third and fourth phase, indicating that almost all vertices could be reconsidered. On the other hand COR and CAS prune the majority of vertices shortly after they become asymmetric, resulting in significantlly smaller queues throughout the later phases.

Overall, the experiments show that for random graphs the update procedure behaves predictably. The queue size typically grows during the middle of the update and then decreases again once most vertices become asymmetric.

6.3.4 Changed vertices in the final coloring

Figure 6.9 shows the number of vertices whose color changes after the update for different perturbation levels α. Since the results for CAS and COR algorithms are very different than the ones for CUP, we do not normalize ΔC by n, but instead group the graphs into size bins.

For CUP, the number of changed vertices grows visibly with both n and α. This illustrates the potential scale of propagation when no pruning is used. However, it is worth noting that even for CUP the update remains smaller than n: not all vertices need to be recolored to compute the new stable partition.

Algorithms CAS and COR give different results. For all values of α, the number of recolored vertices is very small and in a large fraction of runs equal to zero. The distributions are also remarkably stable across perturbation scales. COR performs slightly better than CAS, with fewer nonzero updates and smaller maximum values.

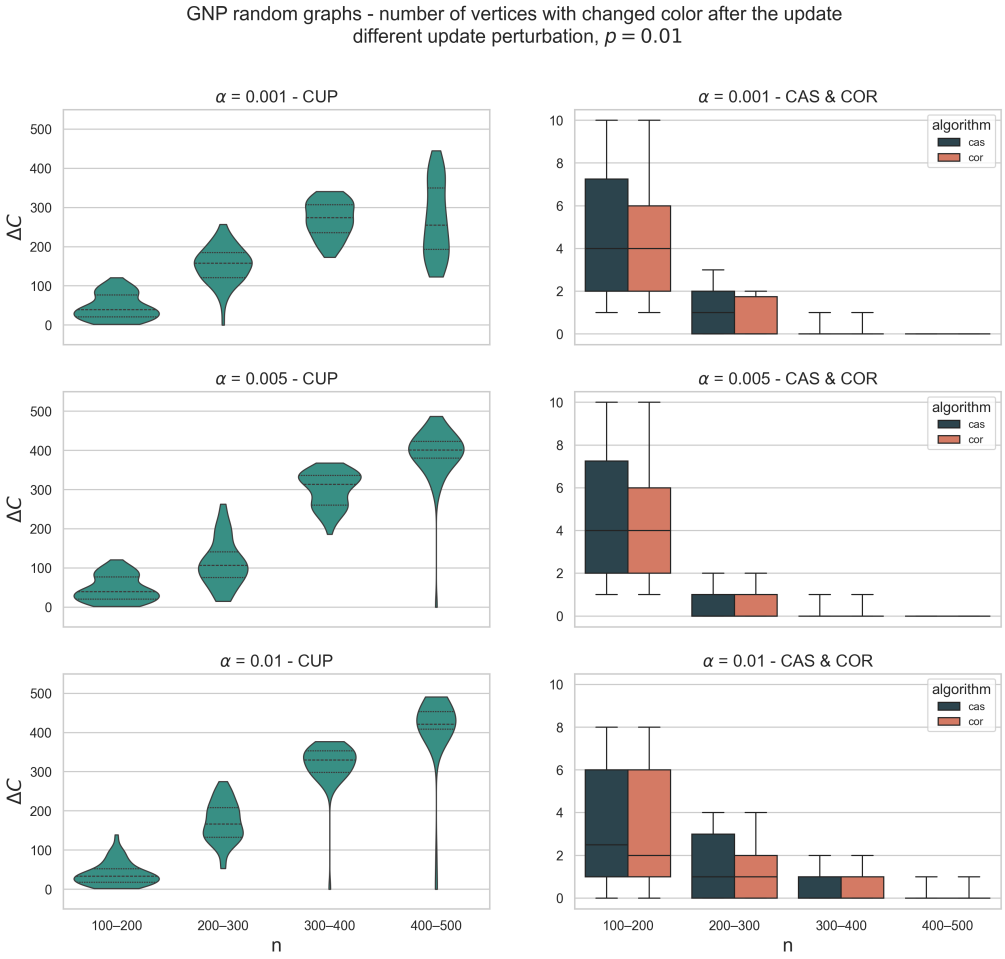


Figure 6.9: Number of vertices whose color changes after the update (ΔC) on $G(n, 0.01)$ graphs for $\alpha \in \{0.001, 0.005, 0.01\}$. The horizontal axis shows binned graph sizes n, and the vertical axis shows ΔC. Left panels display violin plots for CUP (dashed lines indicate median and quartiles). Right panels show boxplots for CAS and COR; the box covers the middle 50% of the data (with the median shown as a central line), and the whiskers range between the 10th and 90th percentiles.

These observations are reflected in Table 6.5. For CAS and COR, the number of changed vertices remains very small in typical cases: in 75% of all runs, at most two vertices are recolored across all values of α.

In contrast, CUP consistently produces much larger updates. The median number of recolored vertices is on the order of 10^2 and increases with α, with very high probability $p(\Delta C > 0) \approx 1$ in all cases.

The probability $p(\Delta C > 0)$ for CAS and COR is ranges roughly between 0.4 and 0.55 depending on α. Thus, while recolorings occur in a substantial fraction of runs, they typically affect only a very small number of vertices.

It is worth noting that the maximum observed change for CAS and COR can be large (up to around 200 vertices), indicating that although recolorings are usually minimal, larger propagation events are still possible in rare cases.

algorithm	median	Q1–Q3	max	$p(\Delta C > 0)$
$\alpha = 0.001$				
cup	169.5	[88, 256.5]	445	0.996
cas	0	[0, 2]	121	0.456
cor	0	[0, 2]	121	0.436
$\alpha = 0.005$				
cup	161.5	[67, 336]	487	0.996
cas	0	[0, 2]	77	0.404
cor	0	[0, 2]	77	0.392
$\alpha = 0.01$				
cup	209.5	[67.25, 358]	491	0.988
cas	1	[0, 2]	199	0.552
cor	0.5	[0, 2]	199	0.500

Table 6.5: Change in the number of recolored vertices ΔC in random graphs for different perturbation levels α. A value $\Delta C = 0$ for CUP occurs when the updated graph requires more refinement iterations than the original one, but the final partition remains unchanged.

To summarise, CAS and COR demonstrate that, in our dataset, a small change in the graph affects only a minor part of the color refinement partition. While CUP illustrates how far a perturbation may propagate, CAS and, in particular, COR keep the final recoloring very small and largely independent of n and α.

6.4 Summary

In all experiments, we applied small edge modifications to different graph families and compared the behavior of the three update procedures. The pruning variants CAS and COR consistently performed well: they reduced the amount of work required after an update, while CUP served as a useful reference for how far a change can propagate without pruning.

The impact of an update depends on both the structure of the graph and the scale of the modification. Sparse graphs and very small perturbations lead to strongly local behaviour, whereas denser graphs or larger changes produce wider queues. Nevertheless, the number of vertices whose color actually changes remains small in almost all cases. The values of ΔC illustrate this clearly and confirm that only limited adjustments are needed to restore the CR partition.

Overall, the results indicate that updating the existing refinement is an effective alternative to recomputing it from scratch. In particular, CAS and COR capture the necessary changes while avoiding most of the additional work.

7 Conclusion and future work

In this thesis we studied the problem of maintaining color refinement under small changes of the input graph. We introduced the dynamic color refinement algorithms: basic update procedure CUP and two pruning variants, CAS and COR, which reduce unnecessary propagation during the update. For each algorithm we analyzed correctness and complexity, and evaluated their behavior experimentally on random trees and random graphs.

The experiments show that in many cases the effect of a local change in the graph remains highly localized. In particular, the number of vertices whose color changes in the final partition is typically very small. The pruning strategies used in CAS and COR significantly reduce the amount of work compared to the basic propagation procedure, while still producing the correct coloring.

The algorithms presented in this thesis open several directions for further research.

Generalization.
The current study focuses on undirected graphs with edge insertions and deletions. A natural extension would be to adapt the algorithms to more general graph models, such as directed graphs or weighted graphs. Another possible direction is to consider updates involving vertex insertions or deletions.

Further complexity analysis.
The thesis provides general bounds on the running time of the proposed algorithms. A more detailed theoretical analysis could investigate the average-case behavior of the algorithms and how it depends on structural properties of the graph.

Memory and data structure optimization.
The algorithms use the history of refinement iterations. Although this appears necessary for the current approach, it would be practical to investigate whether the same functionality could be achieved with less information. This may involve using more efficient data structures and reducing memory usage.

Canonical labeling and symmetry detection.
Color refinement is commonly used as a preprocessing step in graph isomorphism solvers such as NAUTY and TRACES. An important topic connected with this process is looking for canonical labeling, a universal signature for each isomorphic graph. An interesting direction would be to examine if the dynamic update techniques proposed in this thesis could be used to support dynamic symmetry detection or dynamic canonical labeling.

Higher-dimensional Weisfeiler-Leman algorithms.
Color refinement is a the one-dimensional Weisfeiler-Leman algorithm. A natural further question is whether similar update propagation ideas could be extended to higher-dimensional variants of the Weisfeiler-Leman algorithms.

Experiments on larger and real-world datasets.
The experimental evaluation in this thesis focuses on medium-size random trees and random graphs. Further experiments could involve testing the proposed algorithms on larger graphs and on real-world dynamic graph datasets.

Connections to other research fields.

Finally, it would be interesting to explore how the proposed dynamic color refinement algorithms could be applied in areas where color refinement is currently used, such as graph kernels, graph neural networks, and other structural graph representations.

Appendix

Hash-based implementation of color refinement

In the algorithms presented above, colors are determined by color tuples computed by the procedure UPDATE_COLOR_TUPLES. This requires sorting neighbor colors and has complexity

$$O\left(\sum_{v \in q} \deg(v) \log \deg(v)\right).$$

We now describe an alternative implementation that avoids sorting and is more efficient in sparse graphs.

Hashed color refinement

Let the graph be represented by an adjacency matrix A. We use a hashed variant of color refinement based on [22], referred to as HCGCR.

Algorithm 7 HCGCR

Input: adjacency matrix $A \in \{0,1\}^{n \times n}$
Output: stable coloring

Let log_pi be an array where $log_pi(i)$ is the logarithm of the i-th prime
$c^0 \leftarrow \mathbf{1}$
$m_{pre} \leftarrow -1$
$m_{now} \leftarrow 1$
$i \leftarrow 1$
while $m_{pre} \neq m_{now}$ **do**
    $c^i, m \leftarrow \text{COLORS}(c^{i-1} + A \cdot log_pi(c^{i-1}))$
    $m_{pre} \leftarrow m_{now}$
    $m_{now} \leftarrow m$
end while
return c^i

Here, $log_pi(c^{i-1})$ denotes the vector obtained by applying log_pi element-wise to c^{i-1}.

The procedure follows the approach described in [22], where color tuples are encoded using arithmetic operations instead of explicit multiset construction. The correctness of this method, i.e., that equal multisets of neighbor colors lead to identical hash values and thus to the same color class, is discussed in detail in [22].

Algorithm 8 COLORS

Input: array w of size n
Output: labeling c and number of colors

```
M ← empty dictionary
count ← 0
for i = 0 to n − 1 do
    if w[i] ∉ M then
        M[w[i]] ← count
        count ← count + 1
    end if
    c[i] ← M[w[i]]
end for
return c, count
```

The procedure COLORS assigns identical labels to equal hash values, thereby simulating the refinement step.

The refinement history is stored as

$$\text{old_iterations}[i] = \left(c^i_{\text{old}},\ h^i_{\text{old}},\ D^i_{\text{old}}\right),$$

where h^i_{old} denotes the hash values computed in iteration i.

This representation can be used in place of color tuples without changing the structure of CUP or the queue-based update procedures.

Hash-based update

We replace UPDATE_COLOR_TUPLES by the following procedure.

Algorithm 9 UPDATE_HASH_VALUE

Input: A_{upd}, queue $q \subseteq V$, h^i_{old}, c^{i-1}_{upd}
Output: h^i_{upd}

```
h^i_upd ← copy of h^i_old
for all v ∈ q do
    h^i_upd(v) ← c^{i−1}_upd(v) + Σ_{w∈N_upd(v)} log_pi(c^{i−1}_upd(w))
end for
return h^i_upd
```

Complexity

We compare only the cost of recomputing the tuple or hash values for the vertices in the current queue q.

In the original implementation, UPDATE_COLOR_TUPLES recomputes for each vertex $v \in q$ the multiset of colors of its neighbors and sorts it. Therefore, its running time is

$$O\left(\sum_{v \in q} \deg(v) \log \deg(v)\right).$$

In the hash-based implementation, UPDATE_HASH_VALUE replaces sorting by summation. For each vertex $v \in q$, the new hash value is computed by aggregating the transformed colors of its neighbors once. Hence, the recomputation cost is

$$O\left(\sum_{v \in q} \deg(v)\right).$$

Thus, compared with the tuple-based implementation, the hash-based variant removes the logarithmic factor in the local recomputation step.

Use of AI-Tools

AI-based tools (ChatGPT, Gemini) were used during the preparation of this thesis for language refinement and editing. The conceptual work, the development of the algorithms, and the planning and evaluation of the experiments were carried out independently. All content was critically reviewed, adapted, and is the responsibility of the author.

Bibliography

[1] H. Dell, Color refinement visualization tool, https://github.com/holgerdell/color-refinement (2021).

[2] B. Weisfeiler, A. Leman, A reduction of a graph to a canonical form and an algebra arising during this reduction, Nauchno-Tekhnicheskaya Informatsiya 2 (9) (1968) 12–16.

[3] L. Babai, P. Erdös, S. Selkow, Random graph isomorphism, SIAM Journal on Computing 9 (1980) 628–635. doi:10.1137/0209047.

[4] B. D. McKay, A. Piperno, Practical graph isomorphism, ii, Journal of Symbolic Computation 60 (2014) 94–112. doi:10.1016/j.jsc.2013.09.003.

[5] M. Grohe, K. Kersting, M. Mladenov, E. Selman, Dimension reduction via colour refinement, in: Algorithms – ESA 2014, Springer, 2014, pp. 505–516.

[6] N. Shervashidze, P. Schweitzer, E. J. van Leeuwen, K. Mehlhorn, K. Borgwardt, Weisfeiler–lehman graph kernels, Journal of Machine Learning Research 12 (9) (2011).

[7] K. Xu, W. Hu, J. Leskovec, S. Jegelka, How powerful are graph neural networks?, in: International Conference on Learning Representations (ICLR), 2019.

[8] M. Grohe, Word2vec, node2vec, graph2vec, x2vec: Towards a theory of vector embeddings of structured data, in: Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, ACM, 2020, pp. 1–16. doi:10.1145/3375395.3387641.

[9] B. Scheidt, N. Schweikardt, Color refinement for relational structures (2026). arXiv:2407.16022, doi:10.48550/arXiv.2407.16022.

[10] J. Hopcroft, An $n \log n$ algorithm for minimizing states in a finite automaton, in: Theory of Machines and Computations, Elsevier, 1971, pp. 189–196.

[11] R. Paige, R. E. Tarjan, Three partition refinement algorithms, SIAM Journal on Computing 16 (6) (1987) 973–989.

[12] M. Grohe, Color refinement and its applications, in: Inaugural Lecture, 2017, technical report.

[13] P. Sankowski, Faster dynamic matchings and vertex connectivity, in: Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA), 2007, pp. 118–126.

[14] C. Demetrescu, G. F. Italiano, Experimental analysis of dynamic all-pairs shortest path algorithms, ACM Transactions on Algorithms 2 (4) (2006) 578–601. doi:10.1145/1198513.1198519.

[15] P. Sankowski, Subquadratic algorithm for dynamic shortest distances, in: Computing and Combinatorics, Springer, 2005, pp. 461–470.

[16] E. Bergamini, H. Meyerhenke, Fully-dynamic approximation of betweenness centrality, in: Algorithms – ESA 2015, Springer, 2015, pp. 155–166.

[17] K. Hanauer, M. Henzinger, C. Schulz, Recent advances in fully dynamic graph algorithms, arXiv:2102.11169 (2021).
URL https://arxiv.org/abs/2102.11169

[18] D. Eppstein, Z. Galil, G. F. Italiano, Dynamic graph algorithms, in: Algorithms and Theory of Computation Handbook, CRC Press, 1999, Ch. 26.

[19] S. Kiefer, B. D. McKay, The iteration number of colour refinement, arXiv:2005.10182 (2020).
URL https://arxiv.org/abs/2005.10182

[20] N. Shervashidze, P. Schweitzer, E. J. van Leeuwen, K. Mehlhorn, K. M. Borgwardt, Weisfeiler-lehman graph kernels, Journal of Machine Learning Research 12 (77) (2011) 2539–2561.
URL http://jmlr.org/papers/v12/shervashidze11a.html

[21] C. Berkholz, P. Bonsma, M. Grohe, Tight lower and upper bounds for the complexity of canonical colour refinement, Theory of Computing Systems 60 (2017) 581–614. doi:10.1007/s00224-016-9686-0.

[22] K. Kersting, M. Mladenov, R. Garnett, M. Grohe, Power iterated color refinement, in: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 28, 2014. doi:10.1609/aaai.v28i1.8992.

[23] B. Bollobás, Random Graphs, 2nd Edition, Vol. 73 of Cambridge Studies in Advanced Mathematics, Cambridge University Press, 2001. doi:10.1017/CBO9780511814068.