Engineering Hypergraph Node Coloring Algorithms

Julian Franz-Bühling

December 05, 2025

4178864

Master Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University

Supervisor:
Univ.-Prof. PD. Dr. rer. nat. Christian Schulz

Co-Referee:
Prof. Dr. Felix Joos

Co-Supervisor:
Henrik Reinstädtler

Acknowledgments

I would like to express my sincere gratitude to Prof. Dr. Christian Schulz for giving me the opportunity to write this thesis under his supervision. My thanks also go to Henrik Reinstädtler for his continuous help and support throughout the creation of this thesis.

Finally, I would like to thank my wife for her unwavering and enduring support. I dedicate this work to my daughter who was born in April this year.

Abstract

Edge-Colored Clustering (ECC) is a hypergraph partitioning problem where edges are labeled with colors, and the objective is to assign colors to vertices such that an edge is satisfied when all its vertices receive the edge's color. The framework applies to community detection, recommendation systems, and data mining, where heterogeneous relationships influence clustering structure. Although recent work provides theoretical foundations and exact optimization methods, scalable algorithms for large hypergraphs remain limited.

This thesis develops and evaluates three greedy algorithms for node coloring in edge-colored hypergraphs, each tailored to a different optimization objective. We introduce a core-color prioritization strategy that restricts the palette to the k most frequent edge colors, improving scalability by focusing on dominant structural patterns. The resulting algorithms - CoreColor, FairColor, and ProtectColor - address global edge satisfaction minimization, satisfaction across colors, and constraint-based optimization with protected colors.

CoreColor uses vote-weighted greedy initialization with local refinement to minimize total unsatisfied edges. FairColor applies a min-max fairness objective via parameter grid search, and ProtectColor enforces minimum satisfaction guarantees for designated protected colors while optimizing global quality.

Experiments across diverse hypergraph instances show that our greedy algorithms achieve solution quality close to exact Binary Linear Programming (BLP) methods while offering substantial computational advantages. CoreColor matches BLP-level quality but is up to 5 191 times faster and up to 1 563 times more memory-efficient. FairColor provides competitive fairness-aware solutions with speedups up to 203 and memory reductions up to 2 786. ProtectColor further shows that constraint-enforced greedy methods can yield tighter practical protection guarantees than the theoretical worst-case bounds of exact approaches.

Overall, the results show that carefully designed greedy heuristics offer an excellent balance between solution quality and computational efficiency, enabling ECC to scale to instances beyond the reach of exact methods.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Partitioning complex networks into meaningful clusters is a fundamental problem in data analysis and machine learning. In recent years, research has increasingly focused on clustering problems defined over richer data structures that capture multi-relational or categorical information beyond simple pairwise connections. Two prominent directions have emerged: *hypergraph clustering*, which models higher-order (multiway) relationships among entities [37, 38, 28, 32], and *edge-colored clustering*, which captures interactions of multiple types or contexts through edge labels [6, 11, 7, 35, 40, 5].

Edge-colored clustering (ECC) has recently gained attention as a powerful framework for analyzing datasets characterized by categorical or multi-relational interactions [8]. In this framework, data are represented as an *edge-colored hypergraph*, where each hyperedge connects a set of related entities (nodes), and the color of a hyperedge encodes the type, category, or context of the interaction. The central task in ECC is to assign colors to nodes such that the node coloring aligns with the underlying edge colors, thereby capturing consistent and interpretable relationships between node attributes and interaction types.

Hypergraphs provide a natural way to model higher-order interactions that cannot be captured by ordinary graphs [10, 41, 42]. In many real-world systems, relationships occur not between pairs of entities, but among groups, such as research collaborations, co-purchases, biochemical reactions, or team-based tasks. ECC leverages this hypergraph structure to formulate clustering objectives that align node labels with the categorical patterns encoded by edge colors.

ECC has found applications across diverse domains. In *temporal hypergraph clustering* [6], edge colors represent time intervals during which interactions occur, allowing node colors to reveal temporal activity patterns. Variants of ECC have also been used for *team formation* [5], where nodes correspond to individuals, edges represent collaborative tasks, and colors indicate task types. This connects to a broader body of work in sociology, psychology, and organizational behavior [27, 33, 36], where group formation and role assignment have been extensively studied. Other applications include brain network analysis, where edge colors denote co-activation types between brain regions [22], and social or communication networks, where color labels represent interaction categories such as friendship, collaboration, or communication medium [35, 40].

To provide an intuitive example, consider a distributed computing environment in which each node represents a computer and each hyperedge corresponds to a computational task involving multiple machines. Each task (edge) is associated with a color indicating its type, for instance, data preprocessing, model training, or simulation. The goal of ECC is to assign each computer a node color reflecting its primary specialization, ensuring that most tasks are executed by compatible machines while still permitting cross-specialization collaboration where beneficial.

From an algorithmic perspective, such clustering problems are computationally challenging. Even the pairwise case – known as *correlation clustering* [9] – is NP-hard, and by extension, the corresponding ECC formulations on hypergraphs inherit this intractability. Consequently, there is a strong motivation for efficient heuristics that can produce high-quality, interpretable solutions in practice.

In this work, we develop and analyze greedy algorithms for node coloring in edge-colored hypergraphs. Each algorithm corresponds to a distinct coloring strategy that reflects different relationships between node and edge colors, ranging from purely objective-driven heuristics to approaches incorporating fairness and constraint-based considerations.

1.2 Our Contribution

In this thesis, we introduce three greedy algorithms for node coloring in edge-colored hypergraphs, addressing three related problem formulations: Edge-colored Clustering (MinECC), Color-Fair MinECC (CFECC), and Protected-Color MinECC (PCECC). Each algorithm represents a distinct optimization perspective within the ECC framework, balancing global optimality, fairness, and color-specific guarantees.

Our methods are based on greedy heuristics that employ restricted color palettes derived from the frequency distribution of edge colors in real-world hypergraphs. In particular, we introduce a *core-color prioritization* strategy, where the k most frequent edge colors are identified and used as the active color set for node assignments. This restriction not only improves scalability but also aligns the algorithmic focus with the dominant structural patterns present in the data.

We propose three algorithms corresponding to these formulations:

- **CoreColor:** A global optimization heuristic that minimizes the total number of unsatisfied edges across all colors. The method combines vote-weighted greedy color assignment with iterative local refinement to enhance overall solution quality.
- **FairColor:** A fairness-oriented heuristic that iteratively identifies and improves the worst-performing color class-defined as the one with the highest proportion of unsatisfied edges. By focusing local search on this color in each iteration, FAIRCOLOR aims to achieve balanced performance across all colors, reducing the dominance of any single color in the optimization objective.
- **ProtectColor:** A constraint-based formulation that enforces satisfaction guarantees for edges of a designated *protected color* (typically the median-frequency color) while optimizing over the remaining edges. We evaluate a soft-constraint variant that allows a limited fraction of unsatisfied protected edges, enabling a systematic study of the trade-off between color-specific guarantees and overall objective quality.

CORECOLOR and FAIRCOLOR are evaluated across multiple core sizes, while PROTECTCOLOR is additionally tested under varying protection levels. This setup allows a systematic investigation of the trade-offs between palette restriction, constraint satisfaction, runtime, and solution quality across diverse problem instances. To support a broad and realistic experimental analysis, and to ensure comparability with prior work, we extend existing ECC evaluation benchmarks beyond the datasets introduced by Amburg et al. [6] and Veldt et al. [39]. Using recent hypergraph b-matching techniques [30], we generate additional large-scale hypergraph instances, providing deeper empirical insights into algorithmic behavior and scalability.

1.3 Structure

The remainder of this thesis is organized as follows: Chapter 2 introduces the fundamental concepts and general definitions required throughout this work to understand the concepts of hypergraphs and node coloring. It also introduces the three considered problem formulations. Chapter 3 provides an overview of related work and summarizes previous achievements in the fields of ECC and hypergraph analysis. Chapter 4 presents the greedy algorithms developed to address these problems, providing detailed insights into their design and operation. In Chapter 5, we evaluate the proposed methods experimentally under different parameter settings and across a diverse collection of hypergraph datasets. Finally, Chapter 6 concludes the thesis with a discussion of the obtained results and outlines potential directions for future research.

CHAPTER 2

Fundamentals

This chapter introduces the fundamental concepts and notations that are used throughout this thesis. We begin with the general definitions required to formalize the framework of *ECC* in the context of hypergraphs. Afterward, we discuss the three problem variants considered in this work, all of which are formulated within this hypergraph-based setting.

2.1 General Definitions

An instance of *ECC* is represented by a hypergraph

$$H = (V, E, \ell, \omega),$$

where V denotes the set of vertices, E denotes the set of hyperedges (hereafter referred to simply as *edges*), $\ell : E \to [k]$ is a labeling function that assigns each edge a color from the finite color set $[k] = \{1, 2, \ldots, k\}$, and $\omega : E \to \mathbb{R}_{\geq 0}$ is a weight function that assigns each edge a nonnegative weight.

Each edge $e \in E$ may optionally be associated with a weight $\omega(e) > 0$, which reflects the importance or frequency of that edge in the dataset. In the unweighted version of the problem, all edges have unit weight, i.e., $\omega(e) = 1$ for all $e \in E$. For a color $c \in [k]$, we denote by $E_c \subseteq E$ the subset of edges having color c, formally defined as

$$E_c = \{e \in E \mid \ell(e) = c\}.$$

The *rank* of a hypergraph H is defined as the maximum cardinality of any edge, that is,

$$r = \max_{e \in E} |e|.$$

This parameter captures the maximum number of vertices that participate in any single hyperedge. When $r = 2$, the hypergraph reduces to an ordinary graph.

Figure 2.1 illustrates an example of an edge-colored hypergraph with overlapping edges. At this stage, the vertices have not yet been assigned colors, representing the initial state before solving any ECC related problem.

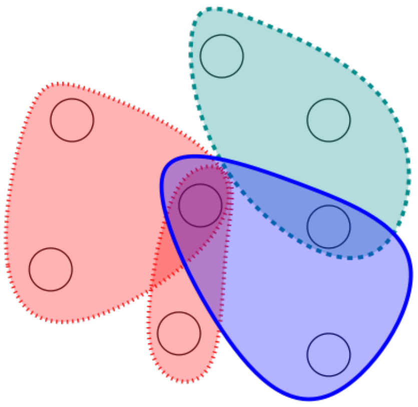


Figure 2.1: Example of an edge-colored hypergraph with overlapping edges. Vertices are not yet colored.

2.2 Problem Definitions

In this section, we formulate the three main node coloring problems for hypergraphs that are relevant for this thesis. Each problem variant addresses different aspects of fairness and optimization in the context of ECC.

MinECC. Let $\lambda : V \to [k]$ be a vertex coloring that assigns each vertex $v \in V$ a color $\lambda(v) \in [k]$. For an edge $e \in E$, we say that e is *satisfied* under coloring λ if all vertices in e have the same color as the edge, that is, if $\lambda(v) = \ell(e)$ for all $v \in e$. Conversely, if at least one vertex in e has a different color than the edge color, we say that e is *unsatisfied*, and the coloring λ incurs a penalty of $\omega(e)$ for this edge.

This formulation can equivalently be viewed as a clustering problem: the vertices are partitioned into k clusters, each corresponding to one color. The objective is to minimize the total weight of edges that do not contain at least one vertex colored with the edge's color. In other words, an optimal coloring aims to align vertex colors with edge colors as consistently as possible.

Formally, the MINECC problem is defined as follows:

MINECC

Input: An edge-colored hypergraph $H = (V, E, \ell, \omega)$ with edge weights $\omega : E \to \mathbb{R}_{\geq 0}$.

Objective: Find a vertex coloring $\lambda : V \to [k]$ that minimizes

$$\sum_{e \in E \text{ unsatisfied}} \omega(e).$$

Figure 2.2 illustrates a scenario where vertices have been colored to maximize the number of satisfied edges (equivalently, minimize the number of unsatisfied edges). As shown

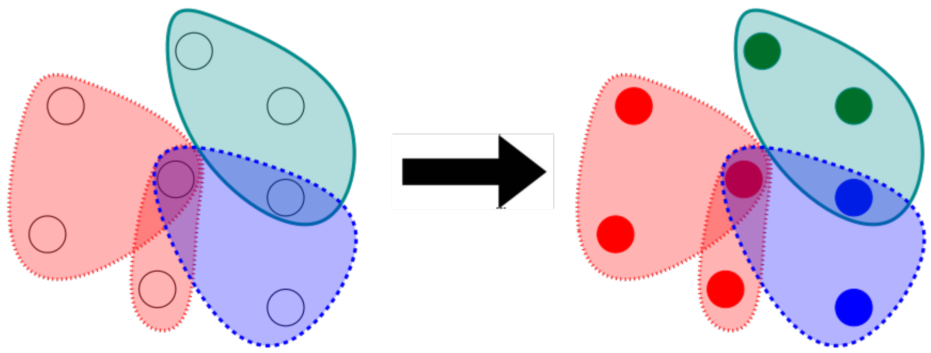


Figure 2.2: Example of a vertex coloring. Three edges (two red and one green) are satisfied, while the blue edge is unsatisfied.

in the figure, the more overlapping a hypergraph is and the more vertices are shared by edges, the more complex the optimization becomes.

CFECC. The MinECC objective may produce solutions that heavily penalize edges of certain colors while favoring others. In applications where fairness across different edge types is important, we require that the number of unsatisfied edges be balanced across all colors. This leads to the CFECC problem.

For a vertex coloring λ, let M_c^λ denote the number (or total weight) of unsatisfied edges of color $c \in [k]$. The goal is to minimize the maximum violation across all colors, thereby ensuring a more equitable distribution of unsatisfied edges.

Color-Fair ECC

Input: An edge-colored hypergraph $H = (V, E, \ell, \omega)$ with edge weights $\omega : E \to \mathbb{R}_{\geq 0}$.

Objective: Find a vertex coloring $\lambda : V \to [k]$ that minimizes

$$\max_{c \in [k]} M_c^\lambda,$$

where $M_c^\lambda = \sum_{\substack{e \in E_c \\ e \text{ unsatisfied}}} \omega(e)$ is the total weight of unsatisfied edges of color c.

This min-max formulation ensures that no single color dominates the set of unsatisfied edges, promoting balanced performance across all edge types. A natural dual formulation is the Color-Fair MaxECC problem, which asks whether there exists a coloring that satisfies at least a specified weight threshold of edges for every color. Although these problems are not equivalent in general, they coincide for hypergraph classes where the number of edges is identical for every color.

PCECC. The CFECC formulation promotes fairness by balancing the number of unsatisfied edges across all colors, preventing any single color from being disproportionately disadvantaged. In certain real-world contexts, however, the focus may lie on safeguarding a specific color that represents a critical or sensitive interaction type, such as connections related to a protected group or priority class.

While it is theoretically possible to satisfy all edges of a chosen protected color c_p by assigning $\lambda(v) = c_p$ for every vertex $v \in V$, such a uniform coloring neglects the structure of other colors and typically yields a poor solution in terms of quality for the original MinECC objective. In practice, we seek a balance between protecting edges of color c_p and maintaining overall clustering quality. This trade-off motivates the following definition:

PROTECTED-COLOR ECC

Input: An edge-colored hypergraph $H = (V, E, \ell, \omega)$ with edge weights $\omega : E \to \mathbb{R}_{\geq 0}$, two integers t and b, and a protected color $c_p \in [k]$.

Objective: Find a vertex coloring $\lambda : V \to [k]$ such that at most t edges are unsatisfied, of which at most b have color c_p.

When $t = b$, we recover the standard MINECC problem. Thus, the protected-color extension can be viewed as a natural generalization that balances global optimization with color-specific protection.

CHAPTER 3

Related Work

ECC has emerged as an active research area at the intersection of machine learning, data mining, and combinatorial optimization. Recent work has established the computational complexity of ECC problems and developed approximation algorithms. However, important open questions still remain, particularly concerning fairness-aware variants and hypergraph extensions.

This chapter surveys existing work on ECC and related problems, examining approximation algorithms, complexity results, and connections to other clustering frameworks. Our review focuses on the recent work of Crane et al. [21], which introduced fairness considerations into ECC and serves as the foundation for this thesis. We identify key gaps in the literature that motivate our contributions and position our work within graph clustering and combinatorial optimization.

3.1 Classical ECC Variants

The study of ECC was initiated by Angel et al. [8] in the context of graphs (i.e., hypergraphs with rank $r = 2$). They introduced the MAXECC problem, which seeks to maximize the number of satisfied edges. Their work established that the problem is polynomial-time solvable for $k = 2$ colors but becomes NP-hard for $k \geq 3$. They also provided the first approximation algorithm for the NP-hard case, achieving a factor of $\frac{1}{e^2} \approx 0.135$ by rounding a linear programming relaxation.

Following this initial work, a sequence of papers progressively improved the approximation guarantees for graph-based MAXECC when $k \geq 3$. Ageev and Kononov [1, 2] developed refined rounding techniques, while Alhamdan and Kononov [3] contributed both improved algorithms and hardness results. The best known approximation factor for MAXECC on graphs currently stands at $\frac{4225}{11664} \approx 0.3622$ [2], and it has been shown that approximating the problem beyond a factor of $\frac{241}{249} \approx 0.972$ is NP-hard when $k \geq 3$ [3].

3.2 Extension to Hypergraphs

More recently, Amburg et al. [6] extended the study of ECC to hypergraphs ($r > 2$) and shifted focus to the MINECC variant, which minimizes the number of unsatisfied edges rather than maximizing satisfied ones. While these two objectives are equivalent at optimality, they differ significantly in terms of approximation guarantees and algorithmic approaches. Amburg et al. [6] provided several algorithms for hypergraph MINECC, including a $\min\left\{2 - \frac{1}{k}, 2 - \frac{1}{r+1}\right\}$-approximation based on LP rounding and combinatorial algorithms with approximation factors scaling linearly in the hypergraph rank r. They established a connection between MINECC and node-weighted multiway cut, allowing LP relaxations and rounding techniques from multiway cut to be adapted to MINECC.

Building on prior work, Veld [39] further improved the approximation factors for rounding the MINECC LP relaxation, advancing them from $\min\left\{2 - \frac{1}{k}, 2 - \frac{1}{r+1}\right\}$ to tighter guarantees $\min\left\{2 - \frac{2}{k}, 2 - \frac{2}{r+1}\right\}$.

3.3 Parameterized Complexity

Complementing the approximation algorithm perspective, several works have studied ECC using parameterized complexity theory. Cai and Leung [13] demonstrated that both MINECC and MAXECC are fixed-parameter tractable (FPT) when parameterized by the solution size for the graph case. More recently, Kellerhals et al. [34] extended these results to hypergraphs, providing improved FPT algorithms and establishing new parameterized hardness results.

3.4 Connections to Other Clustering Problems

ECC is closely related to several other clustering frameworks in the literature. Most notably, it bears strong connections to *chromatic correlation clustering* [12, 11, 7, 35, 40], which generalizes the classical correlation clustering problem to edge-colored graphs. [9] Both frameworks aim to cluster vertices in a manner that respects edge colors while minimizing clustering errors. The objective functions share two common penalty types: both penalize (1) separating adjacent vertices (i.e., placing endpoints of an edge in different clusters), and (2) placing an edge in a cluster whose color differs from the edge's color.

However, there are fundamental differences between the two problems. Chromatic correlation clustering includes an additional penalty for placing non-adjacent vertex pairs within the same cluster. This design choice reflects an underlying assumption that the absence of an edge signals dissimilarity between vertices. Consequently, chromatic correlation clustering solutions may produce multiple distinct clusters of the same color. In contrast, MINECC does not penalize co-clustering non-adjacent vertices, treating non-edges simply as absent or unmeasured relationships rather than explicit dissimilarity signals. This interpretation is particularly natural for large, sparse real-world networks where exhaustive pairwise measurements are infeasible. As a result, ECC produces one cluster per color.

These modeling differences have significant computational implications. Unlike MinECC, which is polynomial-time solvable for $k = 2$ colors, chromatic correlation clustering remains NP-hard even in the single-color case. Recent work has produced improved approximation algorithms for chromatic correlation clustering, with the current best guarantee being a 2.5-approximation [40], though this result does not extend to the general weighted case. Another notable gap is that chromatic correlation clustering has not been studied in the hypergraph setting, whereas multiple results exist for hypergraph MinECC.

The connection established by Amburg et al. [6] between MinECC and node-weighted multiway cut also situates the problem within the broader family of graph partitioning problems [29, 14, 16, 18]. In the node-weighted multiway cut problem (Node-MC), one is given a vertex-weighted graph containing k designated terminal vertices, and the objective is to remove a minimum-weight set of vertices that disconnects all terminals from one another. This problem is approximation-equivalent to the hypergraph multiway cut problem (Hyper-MC) [17], where the input is a hypergraph with k terminals and the goal is to remove a minimum-weight set of hyperedges to separate the terminals.

Both Node-MC and Hyper-MC generalize the classical edge-weighted graph multiway cut problem (Graph-MC) [23], and they are themselves special cases of the submodular multiway partition problem [26]. The best known approximation factor for Node-MC is $2(1 - 1/k)$, which can be achieved by rounding a linear programming relaxation. This same guarantee applies to MinECC via the reduction to Node-MC, though obtaining this bound for the more general submodular multiway partition problem requires solving and rounding a generalized Lovász relaxation [17].

It is worth noting that when $k = 2$, MinECC can be reduced to the minimum s-t cut problem, which is the two-terminal version of Graph-MC and is solvable in polynomial time. However, this special structure does not extend to $k > 2$ colors. Amburg et al. [6] also explored approximating MinECC through a reduction to Graph-MC, but the objectives of the two problems differ by a multiplicative factor of $(r + 1)/2$, limiting the tightness of this approach.

3.5 Fairness and Balance in Clustering

An important limitation of the classical MinECC and MaxECC objectives is that they measure only the total number of (un)satisfied edges, without regard to how these violations are distributed across colors. This can lead to solutions that disproportionately favor certain edge colors while heavily penalizing others. For example, an optimal MinECC solution might satisfy many edges of one color while leaving nearly all edges of another color unsatisfied.

Such imbalanced solutions may be undesirable in applications where different edge colors represent distinct types of interactions or relationships. For instance, in team formation problems where edge colors correspond to different task types, an imbalanced solution might assign workers to only a subset of available tasks, leaving other tasks unattended. This motivates the development of fairness-aware variants of ECC that explicitly balance satisfaction across colors.

The incorporation of fairness constraints into clustering algorithms has been extensively studied in other contexts. Chierichetti et al. [19] introduced the notion of fair clustering with demographic parity constraints, sparking a line of research on fair variants of classic objectives such as k-median and k-center. For a comprehensive survey of fair clustering methods, we refer to Caton and Haas [15]. Despite this progress, fairness considerations have not been systematically explored in the context of ECC until very recently [20], leaving substantial room for further investigation. Early results indicate that incorporating fairness into ECC poses distinct challenges due to color heterogeneity and high-order interactions encoded by hyperedges.

CHAPTER 4

Algorithms for Edge-Color-Constrained Node Coloring

In this chapter, we present three algorithms designed to address the ECC node coloring problems introduced in Chapter 2. Each algorithm tackles a distinct optimization objective, while following a common two-phase framework: a greedy initialization followed by iterative local refinement.

Although all algorithms share a common foundation, they differ in specific strategies and parameters to achieve their respective objectives. We will explain these differences in detail in the following sections.

The algorithms are presented in order of increasing complexity. We begin with the base CoreColor algorithm in Section 4.1, followed by FairColor in Section 4.2, and conclude with ProtectColor in Section 4.3. For each algorithm, we provide detailed pseudocode, explain the key algorithmic components, and analyze the coloring strategies. This structured approach helps illustrate how each extension builds upon and modifies the base framework to achieve its specific objective.

4.1 CoreColor

The CoreColor algorithm serves as the foundation for the two variants presented later. Its objective is to minimize the total number of unsatisfied edges by assigning each node a color from the set of available edge colors, as described in Chapter 2.

CoreColor operates in two main phases: greedy coloring and iterative refinement. To determine the optimal number of core colors, the algorithm systematically evaluates different values of k by running both phases for each configuration and selecting the one that yields the fewest unsatisfied edges. The selected configuration then undergoes additional fine-tuning to further optimize the solution.

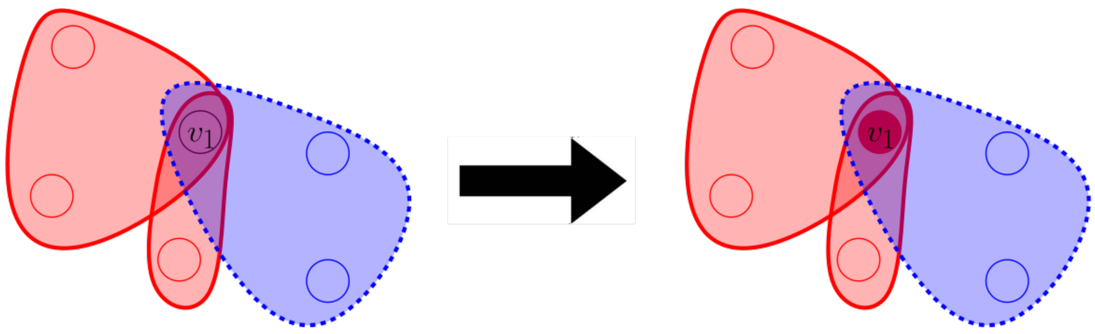


Figure 4.1: Simple example of the greedy coloring process. Node v_1 receives votes from incident edges (2 red, 1 blue) and is colored first due to its high vote diversity and degree. The algorithm assigns red based on the highest vote count.

Greedy Coloring. The greedy initialization constructs an initial node coloring using a voting mechanism where each edge effectively "votes" for its color at each incident node. The process proceeds in three phases:

First, we perform vote counting. For each node v and each edge color c, we count how many edges of color c are incident to v. This count, $\mathit{votes}[v][c]$, represents the potential benefit of assigning color c to node v. A high vote count indicates that coloring v with c could satisfy many edges simultaneously.

Second, rather than coloring nodes in an arbitrary order, we establish a node ordering using a scoring function that combines two factors: color diversity (the number of distinct colors for which node v has received votes, $|\mathit{votes}[v]|$) and node degree (the total number of edges incident to v, $\text{degree}(v)$). The score is computed as $|\mathit{votes}[v]| + \beta \cdot \text{degree}(v)$, where β is a small weight that serves primarily as a tiebreaker. Nodes with higher scores are colored first, as they offer more opportunities for satisfying edges.

Finally, for color assignment, we process each node v in the sorted order and evaluate each candidate color c for which v has received votes. The evaluation considers both votes received (higher vote counts indicate more potential edge satisfactions) and conflicts (neighbors already colored differently from c). We compute a score as $\mathit{votes}[v][c] \cdot \alpha - \mathit{conflicts}$, where the weight α emphasizes satisfying edges over avoiding conflicts. The color with the highest score is assigned to node v.

Algorithm 1 presents the complete greedy coloring procedure, while Figure 4.1 illustrates a simple example of its execution. Node v_1 receives two votes from red edges and one vote from a blue edge, giving it the highest vote diversity and degree in this small graph. According to the node-ordering strategy, v_1 is therefore selected first for coloring. During the color-assignment phase, the algorithm evaluates both candidate colors: red obtains two supporting votes with no conflicts (as no neighbors are colored yet), whereas blue receives only one vote. Following the scoring function $\mathit{votes}[v][c] \cdot \alpha - \mathit{conflicts}$, red achieves the highest score and is consequently assigned to v_1.

Algorithm 1: GreedyColoring (CoreColor)

```
Input  : Hypergraph $H = (V, E)$, set of core colors $C$
Output: Initial node coloring
Initialize nodeColors$[v] \leftarrow -1$ for all $v \in V$;
// Phase 1: Count color votes
foreach edge $e \in E$ do
    if $e$.color $\in C$ then
        foreach node $v \in e$ do
            votes$[v][e$.color$] \leftarrow$ votes$[v][e$.color$] + 1$;
// Phase 2: Order nodes by diversity + degree
foreach node $v \in V$ do
    score$[v] \leftarrow |$votes$[v]| + \beta \cdot$ degree$(v)$
sortedNodes $\leftarrow$ sort $V$ by score (descending);
// Phase 3: Assign colors greedily
foreach node $v$ in sortedNodes do
    bestColor $\leftarrow -1$;
    bestScore $\leftarrow -\infty$;
    foreach color $c \in$ votes$[v]$ do
        conflicts $\leftarrow$ count neighbors with color $\neq c$;
        score $\leftarrow$ votes$[v][c] \cdot \alpha -$ conflicts;
        if score $>$ bestScore then
            bestColor $\leftarrow c$;
            bestScore $\leftarrow$ score;
    nodeColors$[v] \leftarrow$ bestColor;
return nodeColors;
```

Refine Coloring. The iterative refinement phase improves the initial coloring through local search. Starting from the greedy solution, the algorithm repeatedly examines each node to determine whether changing its color would increase the number of satisfied edges. An example is shown in Figure 4.2

Delta Computation. For each node v and candidate color c, we compute the delta, which is the net change in satisfied edges. Each incident edge e contributes $+1$ if it would become satisfied, -1 if it would become unsatisfied, and 0 otherwise.

Algorithm 2 shows the delta computation procedure.

Algorithm 2: ComputeDelta (CoreColor)

```
Input  : Node $v$, new color $newColor$, current coloring $nodeColors$
Output: Change in satisfied edges
1  $delta \leftarrow 0$;
2  foreach edge $e$ incident to $v$ do
3      $wasSatisfied \leftarrow$ all nodes in $e$ have color $e.color$;
4      $wouldBeSatisfied \leftarrow$ true;
5      foreach node $u \in e$ do
6          $c \leftarrow (u = v)\ ?\ newColor : nodeColors[u]$;
7          if $c \neq e.color$ then
8              $wouldBeSatisfied \leftarrow$ false;
9              break;
10     if $wasSatisfied \land \neg wouldBeSatisfied$ then
11         $delta \leftarrow delta - 1$;
12     else if $\neg wasSatisfied \land wouldBeSatisfied$ then
13         $delta \leftarrow delta + 1$;
14 return $delta$;
```

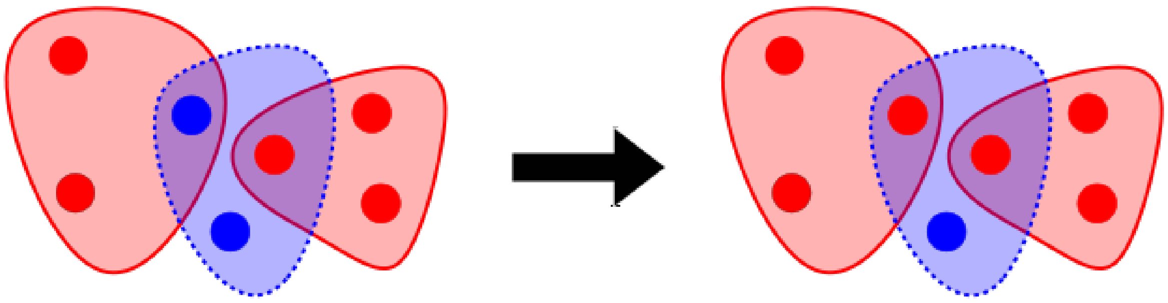


Figure 4.2: Simple example of the local search process. By changing the color of one node from blue to red, we can improve the number of satisfied edges: the red edges increase from one to two, while the blue edge remains unsatisfied.

Refinement Loop. The main refinement procedure runs for a maximum of specified rounds. In each round, nodes are processed in random order to avoid bias. For each node v, we evaluate all core colors by computing their deltas and select the color that yields the maximum improvement. A color change is only applied if it results in a non-negative delta, ensuring the solution never worsens.

The algorithm terminates early if a complete round produces no color changes, indicating that a local optimum has been reached. The randomization of node order helps explore different regions of the solution space across multiple runs.

Complete Algorithm and Parameter Selection. The complete CORECOLOR algorithm systematically searches for the optimal number of core colors to use. Rather than fixing this parameter in advance, CORECOLOR tests values $k = 1, 2, 3, \ldots, k_{\max}$ and selects the configuration that minimizes unsatisfied edges. For each value of k, the algorithm identifies the k most frequently occurring edge colors in the hypergraph. This selection prioritizes colors that appear in many edges, as satisfying these colors has greater impact on the overall objective.

Parallel Evaluation. Since each value of k can be evaluated independently, CORECOLOR runs these evaluations in parallel. For each k, the algorithm executes the greedy coloring followed by iterative refinement, then records the number of unsatisfied edges in the resulting solution. After evaluating all values of k up to $k_{\max}$, CORECOLOR selects the value that produced the fewest unsatisfied edges. This automatic tuning adapts the algorithm to each specific hypergraph structure without requiring manual configuration.

Fine-tuning Phase. After identifying the optimal value of k^*, CORECOLOR performs additional fine-tuning to further minimize unsatisfied edges. The algorithm repeatedly generates new solutions using the selected core colors through greedy coloring and refinement. This process continues until a specified number of consecutive iterations (controlled by the patience parameter p) fail to produce an improvement. The randomization in the refinement phase helps explore different local optima, ensuring the algorithm doesn't settle for the first local optimum found for the best k value.

Algorithm 3 presents the complete CORECOLOR procedure.

Computational Complexity. The greedy phase requires a runtime of $\mathcal{O}(|E| \cdot r_{\max} + |V| \log |V| + |V| \cdot k \cdot d_{\max})$ for vote counting, node sorting, and color assignment, where $r_{\max}$ is the maximum hyperedge size, k is the number of core colors, and $d_{\max}$ is the maximum node degree. Refinement takes $\mathcal{O}(r \cdot |V| \cdot k \cdot d_{\max} \cdot r_{\max})$ time, where r is the number of refinement rounds. With parallel evaluation of $k_{\max}$ core sizes and p fine-tuning iterations, the overall complexity is $\mathcal{O}(k_{\max} \cdot p \cdot (|E| \cdot r_{\max} + r \cdot |V| \cdot k_{\max} \cdot d_{\max} \cdot r_{\max}))$.

Algorithm 3: CoreColor Algorithm

```
Input  : Hypergraph $H = (V, E)$ with edge colors, maximum core size $k_{\max}$,
         refinement rounds $r$, patience parameter $p$
Output: Node coloring minimizing unsatisfied edges
// Parallel evaluation of different core sizes
1  foreach $k \in \{1, 2, \ldots, k_{\max}\}$ in parallel do
2      $C_k \leftarrow$ top $k$ most frequent edge colors in $H$;
3      coloring$_k \leftarrow$ GreedyColoring($H$, $C_k$);
4      RefineColoring($H$, coloring$_k$, $C_k$, $r$);
5      unsatisfied$_k \leftarrow$ count unsatisfied edges in coloring$_k$;
// Select best configuration
6  $k^* \leftarrow \arg\min_k$ unsatisfied$_k$;
7  $C^* \leftarrow$ top $k^*$ most frequent edge colors;
// Fine-tuning phase
8  bestColoring $\leftarrow$ GreedyColoring($H$, $C^*$);
9  RefineColoring($H$, bestColoring, $C^*$, $r$);
10 bestUnsatisfied $\leftarrow$ count unsatisfied edges;
11 noImprovement $\leftarrow 0$;
12 while noImprovement $< p$ do
13     newColoring $\leftarrow$ GreedyColoring($H$, $C^*$);
14     RefineColoring($H$, newColoring, $C^*$, $r$);
15     currentUnsatisfied $\leftarrow$ count unsatisfied edges;
16     if currentUnsatisfied $<$ bestUnsatisfied then
17         bestColoring $\leftarrow$ newColoring;
18         bestUnsatisfied $\leftarrow$ currentUnsatisfied;
19         noImprovement $\leftarrow 0$;
20     else
21         noImprovement $\leftarrow$ noImprovement $+ 1$;
22 return bestColoring;
```

4.2 FairColor

The FAIRCOLOR algorithm addresses a different optimization objective than CORECOLOR. While CORECOLOR minimizes the total number of unsatisfied edges across all colors, FAIRCOLOR employs a min-max strategy that minimizes the maximum number of unsatisfied edges for the colors as formulated in Chapter 2.

Like CORECOLOR, FAIRCOLOR operates in two main phases: greedy coloring and iterative refinement. However, both phases are modified to target the worst-performing color. The algorithm systematically evaluates different parameter combinations to find the optimal balance between vote weighting and worst-color prioritization.

Greedy Coloring. The greedy initialization in FAIRCOLOR extends CORECOLOR'S voting mechanism by incorporating targeted prioritization of the worst-performing color. The core voting and ordering phases remain similar, but the color assignment phase introduces a bonus mechanism.

Vote Counting and Node Ordering. The first two phases follow the same strategy as CORECOLOR. For each node v and each edge color c, we count how many edges of color c are incident to v. This count, $\mathit{votes}[v][c]$, represents the potential benefit of assigning color c to node v. Nodes are then ordered using the scoring function $|\mathit{votes}[v]| + \beta \cdot \text{degree}(v)$, prioritizing nodes with higher color diversity and degree. Nodes with higher scores are colored first.

Color Assignment with Worst-Color Bonus. Before assigning colors, FAIRCOLOR identifies the *worst color* c_w. During color assignment, the scoring function is modified to give priority to the worst color:

$$\mathit{score}(v,c) = \begin{cases} \mathit{votes}[v][c] \cdot \alpha - \mathit{conflicts} + \gamma & \text{if } c = c_w \\ \mathit{votes}[v][c] \cdot \alpha - \mathit{conflicts} & \text{else} \end{cases} \tag{4.1}$$

where α is the vote weight factor (emphasizing edge satisfaction), γ is the bonus weight for the worst color, and *conflicts* counts neighbors already colored differently from c. The bonus weight γ incentivizes the greedy phase to prioritize satisfying edges of the color that currently performs worst.

Algorithm 4 presents the complete greedy coloring procedure for FAIRCOLOR.

Refine Coloring. The iterative refinement phase in FAIRCOLOR differs from CORECOLOR. Rather than optimizing the total number of satisfied edges, FAIRCOLOR'S refinement focuses on reducing the unsatisfied edge count for the worst-performing color.

Algorithm 4: GreedyColoring (FairColor)

```
Input  : Hypergraph $H = (V, E)$, set of core colors $C$, parameters $\alpha$, $\gamma$
Output: Initial node coloring
1  Initialize nodeColors$[v] \leftarrow -1$ for all $v \in V$;
   // Phase 1: Count color votes
2  foreach edge $e \in E$ do
3      if $e$.color $\in C$ then
4          foreach node $v \in e$ do
5              votes$[v][e.$color$] \leftarrow$ votes$[v][e.$color$] + 1$;
   // Phase 2: Order nodes by diversity + degree
6  foreach node $v \in V$ do
7      score$[v] \leftarrow |$votes$[v]| + \beta \cdot \mathrm{degree}(v)$;
8  sortedNodes $\leftarrow$ sort $V$ by score (descending);
   // Identify worst color
9  Update edge satisfaction;
10 $c_w \leftarrow$ color with maximum unsatisfied edges;
   // Phase 3: Assign colors with worst-color bonus
11 foreach node $v$ in sortedNodes do
12     bestColor $\leftarrow -1$;
13     bestScore $\leftarrow -\infty$;
14     foreach color $c \in$ votes$[v]$ do
15         conflicts $\leftarrow$ count neighbors with color $\neq c$;
16         score $\leftarrow$ votes$[v][c] \cdot \alpha -$ conflicts;
17         if $c = c_w$ then
18             score $\leftarrow$ score $+ \gamma$ ;            // bonus for worst color
19         if score $>$ bestScore then
20             bestColor $\leftarrow c$;
21             bestScore $\leftarrow$ score;
22     nodeColors$[v] \leftarrow$ bestColor;
23 return nodeColors;
```

Worst Color Identification. At the start of each refinement round, FAIRCOLOR identifies the current worst color c_t. This color becomes the target for optimization in that round. If all colors have zero unsatisfied edges or the worst color is not in the core set, refinement terminates early.

Algorithm 5: ComputeDeltaForColor (FairColor)

```
Input  : Node $v$, new color newColor, target color $c_t$, current coloring nodeColors
Output: Change in unsatisfied edges for color $c_t$
1  delta ← 0;
2  foreach edge $e$ incident to $v$ where $e.color = c_t$ do
3      wasSatisfied ← all nodes in $e$ have color $e.color$;
4      wouldBeSatisfied ← true;
5      foreach node $u \in e$ do
6          $c$ ← $(u = v)$ ? newColor : nodeColors$[u]$;
7          if $c \neq e.color$ then
8              wouldBeSatisfied ← false;
9              break;
10     if wasSatisfied ∧ ¬wouldBeSatisfied then
11         delta ← delta + 1 ;              // edge becomes unsatisfied
12     else if ¬wasSatisfied ∧ wouldBeSatisfied then
13         delta ← delta − 1 ;                // edge becomes satisfied
14 return delta;
```

Targeted Delta Computation. Unlike CORECOLOR, which computes the change in satisfied edges across all incident edges, FAIRCOLOR computes a targeted delta that considers only edges of the target color c_t:

$$\Delta_{c_t}(v, c_{\text{new}}) = \sum_{\substack{e \in \text{incident}(v) \\ e.\text{color}=c_t}} \begin{cases} -1 & \text{if } e \text{ becomes satisfied} \\ +1 & \text{if } e \text{ becomes unsatisfied} \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

Note the sign convention: a negative delta indicates that the color change would reduce the unsatisfied count for color c_t, which is desirable.

Algorithm 5 shows the targeted delta computation.

Refinement Loop. The main refinement procedure runs for a maximum of specified rounds. At the beginning of each round, the worst color is identified. For each node, the algorithm evaluates candidate colors and selects the one that yields the minimum (most negative) delta for the target color. A color change is only applied if the delta is negative, meaning it reduces the unsatisfied count for the target color.

The algorithm terminates early if a complete round produces no color changes, indicating that a local optimum has been reached for the current worst color. By re-identifying the worst color in each round, CFECC progressively balances the satisfaction across all colors.

Algorithm 6 presents the complete refinement procedure.

Complete Algorithm and Parameter Optimization. The complete FAIRCOLOR algorithm combines worst-color identification, modified greedy coloring, and targeted refinement with parameter optimization through grid search. Like CORECOLOR, for each value of k, FAIRCOLOR identifies the k most frequently occurring edge colors in the hypergraph, prioritizing colors that appear in many edges.

FAIRCOLOR introduces two tunable parameters: the vote weight factor α and the worst-color bonus weight γ. Rather than using fixed values, the algorithm performs a grid search over predefined sets $\alpha \in \mathcal{F}$ and $\gamma \in \mathcal{G}$. For each parameter combination (α, γ), the algorithm runs multiple iterations with early stopping based on a patience parameter p. The algorithm tracks the worst-color unsatisfied count after both greedy and refinement phases, keeping the parameter combination that achieves the minimum worst-color count. This fine-tuning is integrated within the grid search itself, each parameter combination is tested with multiple iterations until no improvement occurs for p consecutive attempts.

Like CORECOLOR, FAIRCOLOR evaluates different values of k in parallel. For each k, the algorithm performs the complete parameter grid search and records the best worst-color unsatisfied count achieved. After all values of k are evaluated, FAIRCOLOR selects the configuration $(k^*, \alpha^*, \gamma^*)$ that minimized the maximum unsatisfied edges for any single color.

Algorithm 7 presents the complete FairColor procedure for a given core size.

Computational Complexity. The greedy and refinement phases have the same complexity as CORECOLOR: $\mathcal{O}(|E| \cdot r_{\max} + |V| \log |V| + r \cdot |V| \cdot k \cdot d_{\max} \cdot r_{\max})$. However, FAIRCOLOR performs a grid search over $|\mathcal{F}| \times |\mathcal{G}|$ parameter combinations with early stopping controlled by patience p. Combined with parallel evaluation of $k_{\max}$ core sizes, the overall complexity is $\mathcal{O}(k_{\max} \cdot |\mathcal{F}| \cdot |\mathcal{G}| \cdot p \cdot (|E| \cdot r_{\max} + r \cdot |V| \cdot k \cdot d_{\max} \cdot r_{\max}))$.

Algorithm 6: RefineColoring (FairColor)

```
Input  : Hypergraph $H = (V, E)$, coloring nodeColors, core colors $C$, max rounds $r$
Output: Improved node coloring
for round $\leftarrow 1$ to $r$ do
    $c_t \leftarrow$ color with maximum unsatisfied edges;
    if $c_t = -1$ or $c_t \notin C$ then
        break ;  // all colors satisfied or worst not in core
    changes $\leftarrow 0$;
    Collect all color changes for this round;
    foreach node $v \in V$ with degree $> 0$ do
        oldColor $\leftarrow$ nodeColors$[v]$;
        bestColor $\leftarrow$ oldColor;
        bestDelta $\leftarrow 0$;
        candidates $\leftarrow \{e.color : e \in \text{incident}(v), e.color \in C\}$;
        foreach color $c \in$ candidates where $c \neq$ oldColor do
            $\Delta \leftarrow$ ComputeDeltaForColor($v$, $c$, $c_t$);
            if $\Delta <$ bestDelta then
                bestDelta $\leftarrow \Delta$;
                bestColor $\leftarrow c$;
        if bestColor $\neq$ oldColor and bestDelta $< 0$ then
            Schedule change: nodeColors$[v] \leftarrow$ bestColor;
            changes $\leftarrow$ changes $+ 1$;
    Apply all scheduled color changes;
    Update edge satisfaction;
    if changes $= 0$ then
        break ;                          // local optimum reached
```

Algorithm 7: FairColor with Core Size k

```
Input  : Hypergraph $H = (V, E)$, core size $k$, refinement rounds $r$, patience $p$,
         factor set $\mathcal{F}$, bonus set $\mathcal{G}$
Output: Best worst-color unsatisfied count and optimal parameters
1  $C_k \leftarrow$ top $k$ most frequent edge colors in $H$;
2  bestWorstCount $\leftarrow \infty$;
3  bestColoring $\leftarrow \emptyset$;
4  bestParams $\leftarrow (\alpha^*, \gamma^*)$;
   // Grid search over parameter combinations
5  foreach $\alpha \in \mathcal{F}$ do
6      foreach $\gamma \in \mathcal{G}$ do
7          noImprovement $\leftarrow 0$;
8          prevWorstCount $\leftarrow \infty$;
           // Iterative optimization with early stopping
9          while noImprovement $< p$ do
10             coloring $\leftarrow$ GreedyColoring($H$, $C_k$, $\alpha$, $\gamma$);
11             worstAfterGreedy $\leftarrow$ worst color unsatisfied count;
12             RefineColoringFairColor($H$, coloring, $C_k$, $r$);
13             worstAfterRefine $\leftarrow$ worst color unsatisfied count;
14             currentWorst $\leftarrow \min($worstAfterGreedy, worstAfterRefine$)$;
15             if currentWorst $<$ bestWorstCount then
16                 bestWorstCount $\leftarrow$ currentWorst;
17                 bestColoring $\leftarrow$ coloring;
18                 bestParams $\leftarrow (\alpha, \gamma)$;
19             if currentWorst $<$ prevWorstCount then
20                 prevWorstCount $\leftarrow$ currentWorst;
21                 noImprovement $\leftarrow 0$;
22             else
23                 noImprovement $\leftarrow$ noImprovement $+ 1$;
24 return (bestWorstCount, bestParams);
```

4.3 ProtectColor

The PROTECTCOLOR algorithm introduces a constraint-based approach to the edge coloring problem. Unlike CORECOLOR, which optimizes all colors uniformly, and FAIRCOLOR, which balances performance across colors, PROTECTCOLOR enforces a soft constraint on a designated *protected color*. It guarantees that at least a specified percentage of edges of this color are satisfied, while still optimizing overall edge satisfaction, as introduced in Chapter 2.

PROTECTCOLOR operates in three main phases: prioritization of the protected color, greedy coloring with constraint awareness, and soft-constraint refinement.

Greedy Coloring. PROTECTCOLOR's greedy phase extends CORECOLOR'S approach by incorporating awareness of the protected color constraint through weighted voting and prioritized initialization.

Phase 0: Protected Color Prioritization. Before the standard greedy phase, PROTECTCOLOR initializes nodes incident to protected edges with the proteced color c_p. Unlike hard constraint mode, these nodes are not locked and can be recolored during subsequent phases. This initialization provides a starting point for satisfying the protected color constraint:

$$\mathit{nodeColors}[v] \leftarrow c_p \quad \forall v \in \bigcup_{\substack{e \in E \\ e.\text{color}=c_p}} e \tag{4.3}$$

Phase 1: Weighted Vote Counting. For all nodes, PROTECTCOLOR computes weighted votes. The weight for each edge e with color c considers the edge size and the number of already-colored nodes:

$$w_e = 1.0 + \frac{1.0}{|e| - n_{\text{colored}}(e) + 1} \tag{4.4}$$

where $|e|$ is the edge size and $n_{\text{colored}}(e)$ is the number of nodes in edge e that already have assigned colors. This weighting scheme prioritizes edges with fewer colored nodes, as they are easier to satisfy.

Each edge then contributes its weight to the vote count:
$\mathit{weightedVotes}[v][c] \leftarrow \mathit{weightedVotes}[v][c] + w_e$ for all nodes v in the edge.

Phase 2: Node Ordering. PROTECTCOLOR orders nodes by computing a priority score:

$$\mathit{priority}(v) = \max_c \mathit{weightedVotes}[v][c] + \beta \cdot \text{degree}(v) \tag{4.5}$$

where the maximum weighted vote represents the strongest color preference for node v. Nodes with higher priority are colored first.

Phase 3: Enhanced Color Assignment. For each node v in priority order, PROTECTCOLOR evaluates each candidate color by considering:

- **Weighted votes**: Higher votes indicate better alignment with edge colors
- **Potential satisfaction**: Count edges that could become satisfied
- **Potential conflicts**: Count edges that would have color conflicts

The scoring function is:

$$\mathit{score}(v, c) = \mathit{weightedVotes}[v][c] + \mathit{potentialSatisfied} \cdot \delta - \mathit{potentialConflicts} \cdot \zeta \tag{4.6}$$

where δ and ζ are weight parameters. An edge is considered potentially satisfiable if it can be satisfied by coloring v with c and at most half of its nodes remain uncolored.

Algorithm 8 presents the complete greedy coloring procedure for PROTECTCOLOR.

Refine Coloring. PROTECTCOLOR employs two distinct refinement strategies applied sequentially, each respecting the soft constraint on the protected color.

Strategy 1: Protected Edge Focus. When the current protected satisfaction falls below the required threshold, PROTECTCOLOR enters a focused mode that prioritizes satisfying protected edges. For each unsatisfied protected edge e, the algorithm attempts to recolor its incident nodes to c_p, accepting changes with delta ≥ -1 (allowing small overall losses to improve protected satisfaction).

This aggressive strategy helps recover constraint satisfaction when the greedy phase or previous refinement has caused violations.

Strategy 2: Constraint-Aware Balanced Optimization. After ensuring the constraint is satisfied, PROTECTCOLOR performs standard local search across all nodes and colors. However, before accepting any color change, the algorithm checks whether it would violate the soft constraint. Specifically, when changing a node colored c_p to a different color, the algorithm:

1. Counts how many protected edges would become unsatisfied
2. Computes the current protected satisfaction count
3. Rejects the change if it would drop protected satisfaction below the threshold

For each node, it evaluates all colors in the allowed set and accepts changes with non-negative delta that maintain the protected color guarantee. This phase balances improvement across all colors while respecting the constraint.

Algorithm 9 presents the soft-constraint refinement procedure.

Complete Algorithm and Constraint Evaluation. The complete PROTECTCOLOR algorithm systematically evaluates the protected color with varying constraint levels to find the optimal balance between protected satisfaction and overall optimization. Rather than fixing a single constraint level, PROTECTCOLOR evaluates multiple allowed error rates $\epsilon \in \{0\%, 5\%, 10\%, \dots, 100\%\}$, creating a constraint relaxation curve that shows how overall satisfaction improves as the protected color constraint is relaxed.

For each constraint level ϵ, PROTECTCOLOR executes the complete algorithm: it collects all colors into the allowed set C, runs greedy coloring with protected color prioritization, applies soft-constraint refinement with the computed mistake budget m, and records both total unsatisfied edges and protected color satisfaction rate. This systematic sweep across constraint levels enables practitioners to select the appropriate balance between protecting critical relationships and optimizing global clustering quality based on application requirements.

Algorithm 10 presents the complete PROTECTCOLOR evaluation procedure.

Computational Complexity. The greedy phase requires a runtime of $\mathcal{O}(|E| \cdot r_{\max} + |V| \cdot |C| \cdot d_{\max} \cdot r_{\max})$, where $|C|$ is the number of allowed colors. The soft-constraint refinement takes $\mathcal{O}(r \cdot (n_p \cdot r_{\max}^2 + |V| \cdot |C| \cdot d_{\max} \cdot r_{\max}))$ time, where n_p is the number of protected edges. Since the algorithm evaluates $|\mathcal{E}|$ constraint levels, the total complexity is $\mathcal{O}(|\mathcal{E}| \cdot (|E| \cdot r_{\max} + r \cdot |V| \cdot |C| \cdot d_{\max} \cdot r_{\max}))$. PROTECTCOLOR has the highest computational cost due to using all colors rather than core colors and evaluating multiple constraint levels.

Algorithm 8: GreedyColoring (ProtectColor)

```
   Input  : Hypergraph $H = (V, E)$, allowed colors $C$, protected color $c_p$
   Output: Initial node coloring
1  Initialize $nodeColors[v] \leftarrow -1$ for all $v \in V$;
   // Phase 0: Prioritize protected color
2  foreach edge $e \in E$ where $e.color = c_p$ do
3      foreach node $v \in e$ where $nodeColors[v] = -1$ do
4          $nodeColors[v] \leftarrow c_p$;
   // Phase 1: Compute weighted votes
5  foreach edge $e \in E$ where $e.color \in C$ do
6      $n_c \leftarrow$ count colored nodes in $e$;
7      $w \leftarrow 1.0 + \frac{1.0}{|e| - n_c + 1}$;
8      foreach node $v \in e$ do
9          $weightedVotes[v][e.color] \leftarrow weightedVotes[v][e.color] + w$;
   // Phase 2: Order nodes by priority
10 foreach node $v \in V$ with degree $> 0$ do
11     $maxVote \leftarrow \max_c weightedVotes[v][c]$;
12     $priority[v] \leftarrow maxVote + \beta \cdot \text{degree}(v)$;
13 $sortedNodes \leftarrow$ sort $V$ by priority (descending);
   // Phase 3: Assign colors with enhanced scoring
14 foreach node $v$ in $sortedNodes$ do
15     $bestColor \leftarrow -1$;
16     $bestScore \leftarrow -\infty$;
17     foreach color $c \in C$ do
18         $score \leftarrow weightedVotes[v][c]$;
19         $potentialSat \leftarrow 0, potentialConf \leftarrow 0$;
20         foreach edge $e \in incident(v)$ where $e.color = c$ do
21             $canSatisfy \leftarrow$ true, $uncoloredCount \leftarrow 0$;
22             foreach node $u \in e$ do
23                 if $nodeColors[u] = -1$ then
24                     $uncoloredCount \leftarrow uncoloredCount + 1$;
25                 else if $nodeColors[u] \neq c$ then
26                     $canSatisfy \leftarrow$ false;
27                     $potentialConf \leftarrow potentialConf + 1$;
28             if $canSatisfy$ and $uncoloredCount \leq |e|/2$ then
29                 $potentialSat \leftarrow potentialSat + 1$;
30         $score \leftarrow score + potentialSat \cdot \delta - potentialConf \cdot \zeta$;
31         if $score > bestScore$ then
32             $bestColor \leftarrow c$;
33             $bestScore \leftarrow score$;
34     $nodeColors[v] \leftarrow bestColor$;
35 return $nodeColors$;
```

Algorithm 9: RefineColoringProtectColor (Soft Constraint)

Input : Hypergraph $H = (V, E)$, coloring *nodeColors*, allowed colors C, protected color c_p, allowed mistakes m, max rounds r

Output: Improved node coloring

```
1  $protectedEdges \leftarrow \{e \in E : e.color = c_p\}$;
2  $requiredSatisfied \leftarrow |protectedEdges| - m$;
3  for $round \leftarrow 1$ to $r$ do
4      $changes \leftarrow 0$;
5      $currentProtectedSat \leftarrow$ count satisfied edges in $protectedEdges$;
       // Strategy 1: Focus on protected edges if below
          threshold
6      if $currentProtectedSat < requiredSatisfied$ then
7          foreach edge $e \in protectedEdges$ where $e$ is unsatisfied do
8              foreach node $v \in e$ where $nodeColors[v] \neq c_p$ do
9                  $\Delta \leftarrow$ ComputeDelta($v$, $c_p$);
10                 if $\Delta \geq -1$ then
11                     $nodeColors[v] \leftarrow c_p$;
12                     Update edge satisfaction for incident edges of $v$;
13                     $changes \leftarrow changes + 1$;
       // Strategy 2: Constraint-aware balanced optimization
14     Shuffle nodes randomly;
15     foreach node $v \in V$ with degree $> 0$ do
16         $oldColor \leftarrow nodeColors[v]$;
17         $bestColor \leftarrow oldColor$;
18         $bestDelta \leftarrow 0$;
19         foreach color $c \in C$ where $c \neq oldColor$ do
20             $\Delta \leftarrow$ ComputeDelta($v$, $c$);
               // Check constraint violation for protected color
                  changes
21             if $c \neq c_p$ and $oldColor = c_p$ then
22                 $wouldUnsatisfy \leftarrow$ count protected edges that would break;
23                 $currentSat \leftarrow$ count satisfied edges in $protectedEdges$;
24                 if $currentSat - wouldUnsatisfy < requiredSatisfied$ then
25                     continue ;                // would violate constraint
26             if $\Delta > bestDelta$ then
27                 $bestDelta \leftarrow \Delta$;
28                 $bestColor \leftarrow c$;
29         if $bestColor \neq oldColor$ and $bestDelta \geq 0$ then
30             $nodeColors[v] \leftarrow bestColor$;
31             Update edge satisfaction for incident edges of $v$;
32             $changes \leftarrow changes + 1$;
33     if $changes = 0$ then
34         break ;                              // local optimum reached
```

Algorithm 10: ProtectColor Constraint Sweep

```
Input  : Hypergraph $H = (V, E)$, refinement rounds $r$, constraint levels $\mathcal{E}$
Output: Results for each constraint level
// Select protected color using median rank
1  Rank all colors by frequency (descending);
2  $c_p \leftarrow$ color at median rank;
3  $n_p \leftarrow |\{e \in E : e.color = c_p\}|$ ;              // count protected edges
4  $C \leftarrow$ all distinct edge colors in $H$;
5  $results \leftarrow []$;
// Sweep through constraint levels
6  foreach $\epsilon \in \mathcal{E}$ do
7      $m \leftarrow \lfloor n_p \cdot \epsilon \rfloor$ ;                   // compute allowed mistakes
       // Run ProtectColor with this constraint level
8      $coloring \leftarrow$ GreedyColoring($H$, $C$, $c_p$);
9      RefineColoringProtectColor($H$, $coloring$, $C$, $c_p$, $m$, $r$);
       // Evaluate results
10     $totalUnsatisfied \leftarrow$ count unsatisfied edges;
11     $protectedSat \leftarrow$ count satisfied edges where color $= c_p$;
12     $protectedRatio \leftarrow protectedSat / n_p$;
13     Store $(\epsilon, totalUnsatisfied, protectedRatio)$ in $results$;
14 return $results$;
```

CHAPTER 5

Experimental Evaluation

After describing our three algorithms in Chapter 4, we now provide an experimental evaluation. We begin by introducing our experimental methodology and the hypergraph instances used in our study. Then, we present the experimental results for all three algorithms.

5.1 Methodology

This section describes the experimental setup, implementation details, baseline comparisons, and evaluation metrics used to assess the performance of our proposed algorithms.

Implementation and Experimental Setup. All algorithms presented in this thesis, CORECOLOR, FAIRCOLOR, and PROTECTCOLOR, are implemented in C++17 and compiled with `g++` 9.4.0 using full optimization via the `-O3` flag. All experiments are conducted on a sixteen-core Intel Xeon Silver 4216 processor running at 2,1 GHz, equipped with 97 GB of main memory, 16 MB of L2 cache, and Ubuntu 20.04.1 as the operating system.

For each algorithm and each instance, we perform ten independent repetitions to ensure statistically robust results. Memory consumption is measured using the system tool `/usr/bin/time -v`, which provides reliable peak memory usage information.

To ensure fair comparison and practical relevance, we impose a timeout of 30 minutes for each graph instance. This timeout excludes input/output operations and focuses solely on algorithm execution time. Instances that exceed this time limit are excluded from the main experimental results but are documented separately for completeness.

Baseline Algorithms. We compare our greedy algorithms against the Binary Linear Programming (BLP) approaches proposed by Crane et al. [21], which address the same optimization problems using exact optimization techniques. These BLP formulations are implemented using the Gurobi optimizer [31], a state-of-the-art commercial solver for mixedinteger programming, which can also utilize multiple cores. The comparison allows us to assess the trade-off between solution quality and computational efficiency.

Performance Profiles. For CORECOLOR and FAIRCOLOR, we employ, among others, *performance profiles* as proposed by Dolan and Moré [25] to compare algorithms across multiple objectives, including solution quality, runtime, and memory consumption.

Performance profiles visualize algorithm performance across multiple instances in a two-dimensional plot. The x-axis represents a performance factor $\tau \geq 1$, while the y-axis shows the fraction of instances (expressed as a percentage). For each algorithm A, a point (f, τ) on its curve indicates that algorithm A performs within a factor of τ of the best algorithm on a fraction f of all instances. More formally, for each instance, we compute the ratio between the metric value of algorithm A and the best metric value achieved by any algorithm on that instance. The curve then shows the cumulative distribution of these ratios across all instances. Algorithms whose curves rise quickly and reach higher y-values are preferable, as they achieve near-optimal performance on a larger fraction of instances.

ProtectColor Evaluation. For PROTECTCOLOR, we evaluate two complementary metrics across varying constraint levels:

- **Protected Color Constraint Satisfaction:** The percentage of edges with the protected color that are satisfied, measuring how well the algorithm respects the soft constraint.
- **Overall Edge Satisfaction:** The total percentage of satisfied edges across all colors, measuring global solution quality.

By sweeping through constraint levels $\epsilon \in \{0\%, 5\%, 10\%, \ldots, 100\%\}$, we construct a trade-off curve that illustrates how enforcing stricter protection for one color affects overall solution quality.

5.2 Datasets

Our experimental evaluation uses two categories of hypergraph instances to provide broad and representative insights into algorithm performance across diverse structural settings.

Benchmark Instances. We first evaluate performance on the standard ECC benchmark suite introduced by Amburg et al. [6] and Veldt et al. [39]. This suite comprises six diverse real-world hypergraphs with varying sizes, edge densities, and color distributions, as shown in Table 5.1. These instances represent established testbeds for ECC algorithms and enable direct comparison with prior work.

Dataset	**|V|**	**|E|**	**r**	**k**
Brain	638	21180	2	2
Cooking	6714	39774	65	20
DAWN	2109	87104	22	10
MAG-10	80198	51889	25	10
Walmart-Trips	88837	65898	25	44
Trivago-Clickout	207974	247362	85	55

Table 5.1: Benchmark datasets for ECC evaluation. $|V|$ denotes the number of vertices, $|E|$ the number of edges, r the maximum edge rank, and k the number of distinct edge colors.

Generated Instances. To gain deeper insights into algorithm behavior on larger and more varied graph structures, we construct additional hypergraph instances using the b-matching algorithm proposed by Grossmann et al. [30]. This technique transforms existing graph datasets into hypergraphs through an iterative decomposition process. In each iteration, a capacity-constrained matching is computed using a greedy selection strategy followed by iterated local search refinement. All edges in this matching are assigned a new color and then removed from consideration. The process repeats with the remaining edges until all edges have been assigned colors. Each hyperedge color thus corresponds to one b-matching layer, and the sequence of iterations yields a complete edge-coloring of the input instance.

We apply the b-matching procedure to two graph collections. The first set is derived from the University of Florida Sparse Matrix Collection [24], a widely used repository containing diverse sparse matrices from scientific computing and network analysis applications. Table 5.2 lists the instances generated from this collection.

The second set is derived from the ISPD98 Circuit Benchmark Suite [4], a standard benchmark from VLSI circuit design. Table 5.3 lists the instances from this suite.

To evaluate scalability on very large instances, we additionally generate three hypergraphs with substantially larger sizes, detailed in Table 5.4.

Dataset	|V|	|E|	r	k
192bit	13093	13691	16	2301
ABACUS_shell_hd	12752	23412	15	7
airfoil_2d	23412	19584	134	23
Andrews	60000	60000	36	6
appu	14000	14000	294	51
as-22july06	22963	22963	2390	342
as-caida	26475	26475	2628	376
astro-ph	16046	16046	360	122
av41092	41092	41,092	2135	240
bayer04	20545	20545	34	18
bcsstk29	13992	13992	71	30
deltaX	21961	68600	83	15
epb1	14734	14734	7	5
EternityII_A	150638	7362	780	2
ex19	12005	12005	50	23
fd18	16428	16428	6	3
finan512	74752	74752	55	20
foldoc	13309	13356	102	240
Franz11	30144	47104	7	11
G2_circuit	150102	150102	6	3
G67	10000	10000	4	2
g7jac040sc	11790	11790	120	34
garon2	13535	13535	45	20
gemat1	10595	4929	4928	10
graphics	11822	29493	4	29
gyro	17361	17361	360	121
hvdc1	24842	24842	181	14
IG5-17	14060	30162	120	406
Ill_Stokes	20896	20896	12	7
image-interp	120000	232485	5	4
language	399130	399130	11555	44
lhr14	14270	14270	63	17
light_in_tissue	29282	29282	18	8
lp_nug20	72600	15240	20	15
lung2	109460	109460	8	4
Maragal_6	10144	21251	5941	247
mri1	114637	65536	9	83
msc10848	10848	10848	723	243
nopoly	10774	10774	11	4
NotreDame_actors	127823	383640	646	98
obstclae	40000	40000	5	3
opt1	15449	15449	243	121
Oregon-1	11174	11174	2389	797
p2p-Gnutella25	22352	6221	64	12
Pd_rhs	5804	4371	14	10
pesa	11738	11738	10	4
PGPgiantcompo	10680	10680	205	69
poli3	16955	16955	336	14
powersim	15838	15838	40	14
psse2	11028	28634	28	26

Table 5.2: Generated datasets from the University of Florida Sparse Matrix Collection.

Dataset	|V|	|E|	r	k
ISPD98_ibm01	12752	14111	42	13
ISPD98_ibm02	19601	19584	134	23
ISPD98_ibm03	23136	27401	55	34
ISPD98_ibm04	27507	31970	46	176
ISPD98_ibm05	29347	28446	17	5
ISPD98_ibm06	32498	34826	35	31
ISPD98_ibm07	45926	48117	25	33
ISPD98_ibm08	51309	50513	75	389
ISPD98_ibm09	53395	60902	39	58
ISPD98_ibm10	69429	75196	41	46
ISPD98_ibm11	70558	81454	24	58
ISPD98_ibm12	71076	77240	28	158
ISPD98_ibm13	84199	99666	24	60
ISPD98_ibm14	147605	152772	33	90
ISPD98_ibm15	161570	186608	36	102
ISPD98_ibm16	183484	190048	40	60
ISPD98_ibm17	185495	189581	36	27
ISPD98_ibm18	210613	201920	66	33

Table 5.3: Generated datasets instances from the ISPD98 Circuit Benchmark Suite.

Dataset	|V|	|E|	r	k
Maggeo	1256385	1590335	25	206
Amazon	2268231	4285363	9350	29
StackOverflow	2675955	11305343	25	101

Table 5.4: Large-scale hypergraph instances.

5.3 MinECC

In this section, we present the experimental results for MinECC. We begin with parameter tuning to determine the optimal values for the CoreColor algorithm, and then we present the results of the experiments.

Parameter Tuning. To determine optimal parameter values for CoreColor, we conduct systematic sensitivity analysis on four key parameters: the vote weight factor α, the number of refinement rounds r, the early stopping patience p, and the maximum core size $k_{\max}$. We evaluate these parameters on seven representative instances from our collection: MAG, Trivago, Dawn, appu, Ill_Stokes, SPD98_ibm11, and PGPgiantcompo. Each parameter is tuned independently while holding others fixed at default values, allowing us to assess the individual impact of each parameter on solution quality. As β serves only as a tiebreaker, we keep it fixed without further tuning.

Vote Weight Factor. We begin by tuning α, which controls the trade-off between maximizing edge color votes and minimizing conflicts during greedy node coloring. For this we fix $r = 1$, $p = 1$, and $k_{\max} = 55$, testing $\alpha \in \{1.0, 1.5, 2.0, 3.0, 5.0, 10.0\}$.

Figure 5.1 presents the results, showing the percentage deviation from the best solution for each dataset across different α values. The results demonstrate that α has some influence on solution quality: some graphs show improvement for certain α values, while other instances remain relatively stable. Setting $\alpha = 2.0$ yields the best overall performance: lower values, such as 1, produce worse solutions, while higher values, such as 10, show similar results. We therefore fix $\alpha = 2.0$ for the final experiments.

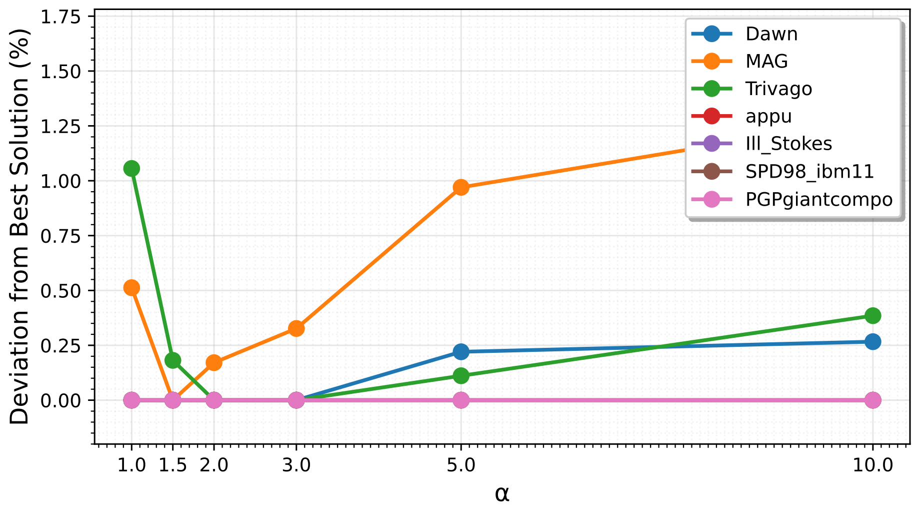


Figure 5.1: Parameter sensitivity for vote weight factor α.

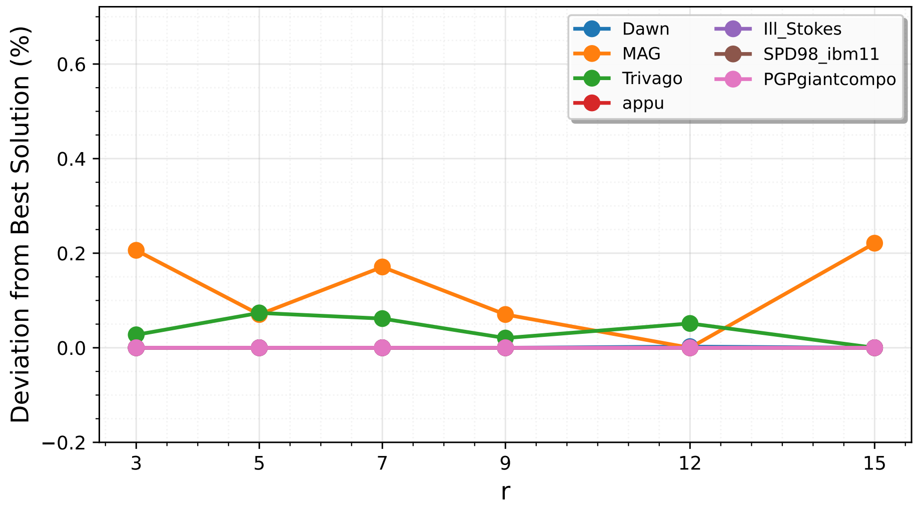


Figure 5.2: Parameter sensitivity for refinement rounds r.

Refinement Rounds. With α fixed at 2.0, we tune the number of local refinement rounds $r \in \{3, 5, 7, 9, 12, 15\}$, holding $p = 1$ and $k_{\max} = 55$ constant.

Figure 5.2 shows that refinement rounds have measurable impact on solution quality for some instances, though the effect varies considerably across datasets. The maximum deviation from optimal reaches approximately 20%, indicating that iterative refinement provides improvements for some graph structures while having minimal effect on others. The value $r = 12$ achieves the best aggregate performance across all instances. We adopt $r = 12$ for the final algorithm configuration.

Early Stopping Patience. The parameter p controls the early stopping criterion: the algorithm continues generating new solutions until p consecutive iterations fail to improve the best result. We test $p \in \{2, 3, 4, 5, 7, 10\}$ with $\alpha = 2.0$, $r = 12$, and $k_{\max} = 55$.

Figure 5.3 reveals similar behavior to the previous parameters. Several instances benefit from increased patience, though not uniformly. The value $p = 3$ produces the best overall solutions across the test suite. Like the refinement rounds parameter, early stopping patience shows moderate influence, with the worst solutions deviating by approximately 20% from optimal. We set $p = 3$ for the final experiments.

Maximum Core Size. Finally, we tune $k_{\max}$, which limits the search space by considering to the top $k_{\max}$ highest-frequency colors as candidates for the core color set. We test $k_{\max} \in \{5, 10, 20, 30, 40, 55\}$ using the previously values $\alpha = 2.0$, $r = 12$, and $p = 3$.

Figure 5.4 demonstrates that $k_{\max}$ exhibits similar sensitivity patterns to other parameters, with varying impact across instances. Notably, Trivago shows substantial improvement with larger $k_{\max}$ values, suggesting that restricting the core color set too aggres-

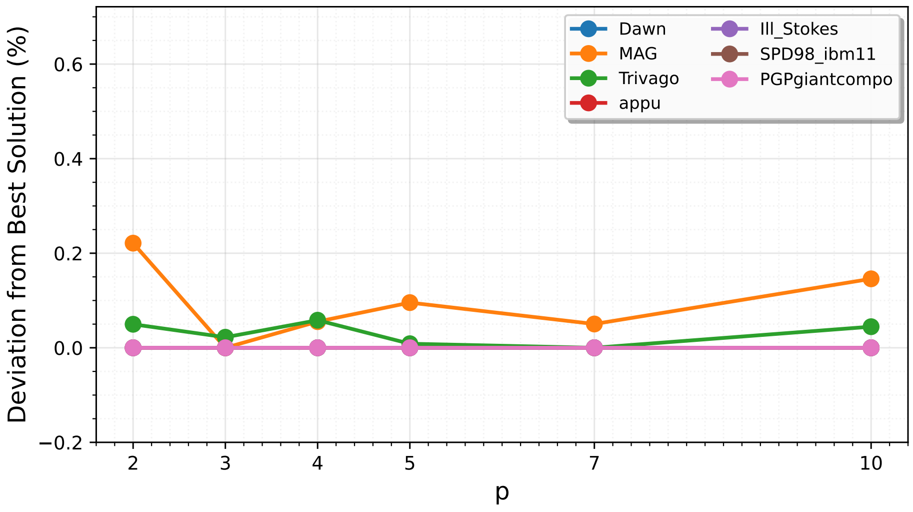


Figure 5.3: Parameter sensitivity for early stopping patience p.

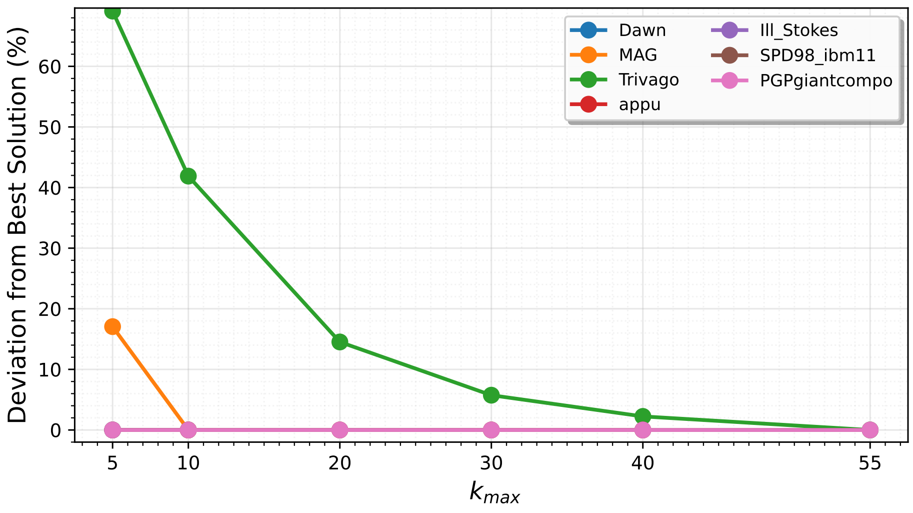


Figure 5.4: Parameter sensitivity for maximum core size $k_{\max}$.

sively can severely limit solution quality for certain graph structures. Conversely, other instances achieve optimal solutions even with small $k_{\max}$ values. To ensure robust performance across diverse instances we retain $k_{\max} = 55$ for our final experiments.

After tuning the parameters, we conduct the final experiments using the following settings: $\alpha = 2$, $r = 12$, $p = 3$, $k_{\max} = 55$, and $\beta = 0.01$. We evaluate CoreColor in comparison with the BLP formulations for MinECC and Color-Fair MinECC proposed by Crane et al. [21]. Our evaluation examines solution quality, runtime, and memory consumption across both benchmark datasets and generated instances. For clarity, we present the results for benchmark datasets and generated instances separately. We begin by discussing the benchmark datasets.

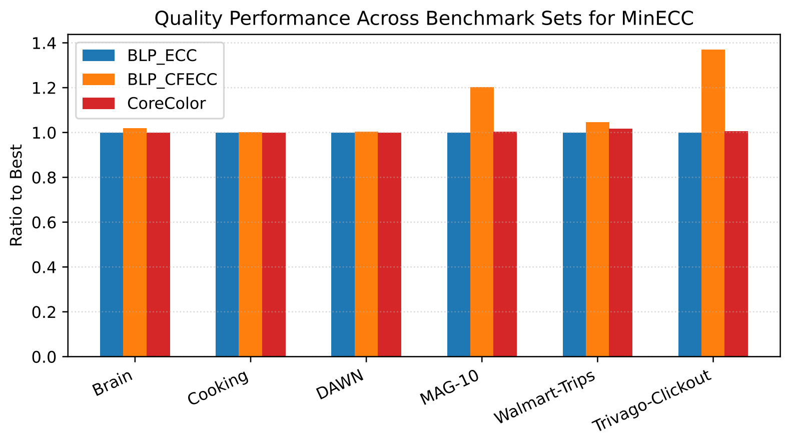
Solution Quality on Benchmark Sets. Figure 5.5a presents the quality profile comparing CoreColor against both BLP formulations. The results demonstrate that CoreColor achieves solution quality nearly identical to BLP_ECC across all benchmark instances. On the most challenging instance, Trivago-Clickout, CoreColor produces solutions within a factor of $1{,}017$ of BLP_ECC, meaning it is at most 1,7% worse than the optimal solution even in the worst case.

In contrast, BLP_CFECC exhibits noticeably lower solution quality on several instances. On MAG-10, it produces solutions with a factor of $1{,}2$ relative to the best result, and on Trivago-Clickout, it performs a factor of $1{,}37$ worse than the optimal solution.

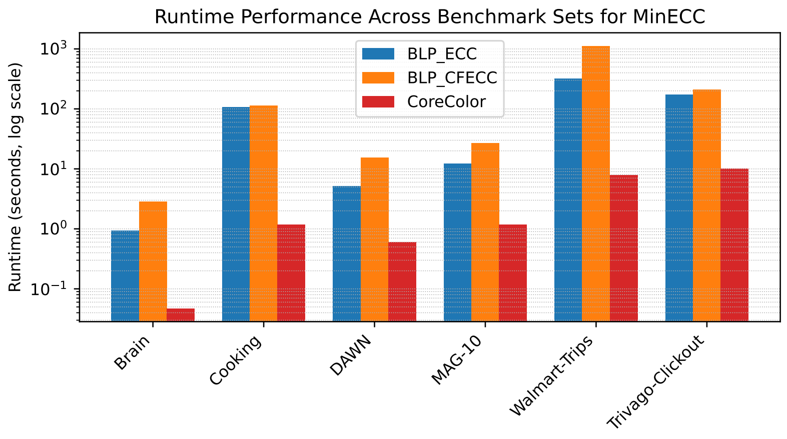
Runtime on Benchmark Sets. Figure 5.5b reveals that CoreColor outperforms both BLP formulations in terms of computational efficiency, consistently delivering the fastest solutions across all benchmark instances.

The runtime advantages are substantial: BLP_ECC requires a factor of 91,2 more time than CoreColor to solve the Cooking dataset, while even in its best case (DAWN dataset), it still needs a factor of 8,7 more time. BLP_CFECC exhibits even greater computational overhead, requiring a factor of 142,5 more time than CoreColor on Walmart-Trips. Even in its best case (Trivago-Clickout), BLP_CFECC still needs a factor of 20,7 more time than CoreColor. These speedups come from the fundamental algorithmic differences: while BLP must explore an exponentially large solution space through branch-and-bound, the greedy approach makes a single deterministic pass followed by bounded local refinement.

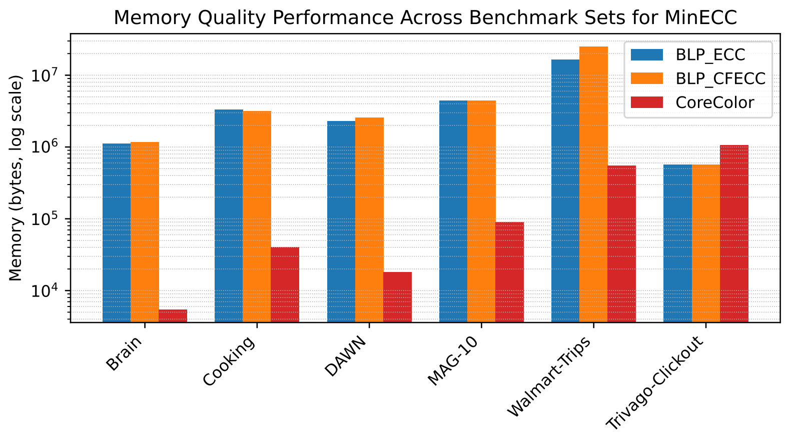
Memory Consumption on Benchmark Sets. The results are shown in Figure 5.5c and highlight similar advantages for CoreColor on most instances. Except for the Trivago-Clickout dataset, CoreColor outperforms the other algorithms by a large margin. For Trivago-Clickout, it requires up to a factor of 1,9 more memory than both BLP formulations. In contrast, the BLP solutions perform quite similarly. BLP_ECC uses a factor of 204,1 more memory than CoreColor for the Brain dataset, while BLP_CFECC uses a factor of 214,1 more memory on the same dataset. This overhead stems from BLP's requirement to maintain explicit binary variables for all node-color assignments, constraint matrices, and branch-and-bound search structures, whereas CoreColor stores only the current coloring.



(a) Solution quality.



(b) Runtime.



(c) Peak memory consumption.

Figure 5.5: Comparison of CoreColor against BLP_ECC and BLP_CFECC across benchmark instances. Each subfigure shows a different metric.

We now present the results for the generated instances. We again compare solution quality, runtime, and memory consumption, using performance profiles to do so.

Solution Quality on Generated Sets. Figure 5.6a presents the quality profile comparing CoreColor against both BLP formulations. The results demonstrate that CoreColor achieves solution quality nearly identical to BLP_ECC, similar to the benchmark instances.

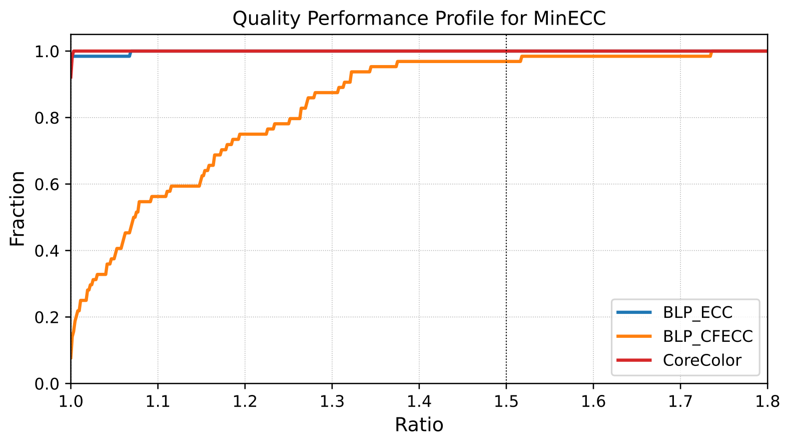
CoreColor starts at a fraction of 0,922, whereas BLP_ECC begins at 0,953. CoreColor reaches full coverage (a fraction of 1.0) at a factor of 1,002, while BLP_ECC achieves the same with a factor of 1,069. BLP_CFECC performs worse, starting at a fraction of 0,078. Half of the instances complete within a factor of 1,073, with the worst-case factor being 1,735.

Runtime on Generated Sets. Figure 5.6b shows that CoreColor behaves similarly to the benchmark sets. It outperforms both BLP formulations in terms of computational efficiency, consistently delivering the fastest solutions across all instances.

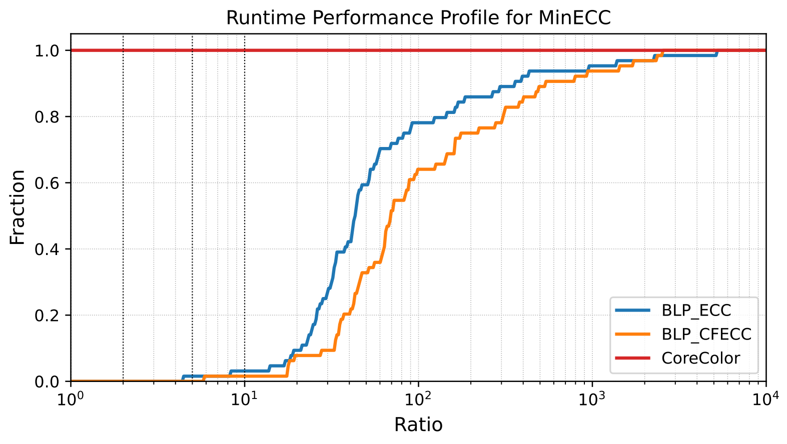
BLP_ECC first exceeds zero performance coverage at $\tau \approx 4{,}46$, indicating that it requires at least this factor before solving any instance. For BLP_ECC, 50% of the instances take up to a factor of 42 longer to solve than CoreColor, with a worst-case factor of 5 191. Similarly, BLP_CFECC first exceeds zero at $\tau \approx 5{,}88$, showing an even slower start. It exhibits a median factor of 68 and a worst-case factor of 2 541. As demonstrated above, the BLP formulations require substantially more time and memory than our greedy algorithm. Several instances exceed the 30-minute timeout when solved using the BLP formulations. A complete overview of all results, including datasets marked as out-of-time (OOT), is provided in Section A.1.

Memory Consumption on Generated Sets. Figure 5.6c shows that CoreColor consistently requires the least memory across all instances.

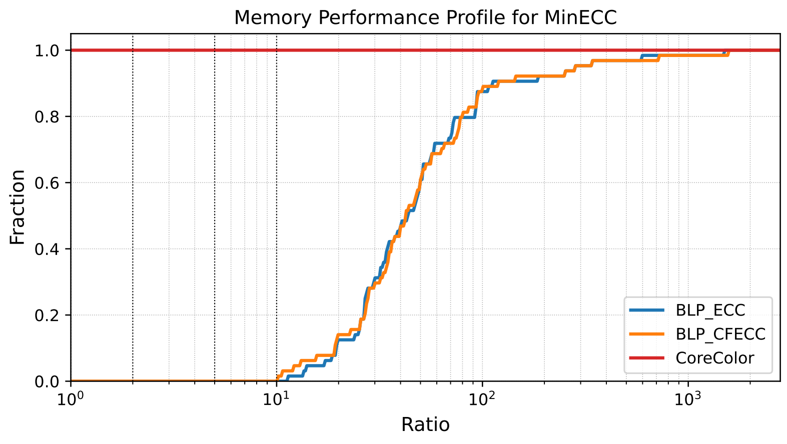
Both BLP formulations begin to rise noticeably only at a factor of around 10, indicating substantially higher memory demands. Compared to BLP_ECC, CoreColor requires significantly less memory, with a median reduction factor of 43,30 and a worst-case reduction factor of 1 492,15. Against BLP_CFECC, the memory savings are comparable, with a median reduction factor of 42,41 and a worst-case factor of 1 563,90. For the large-scale instances listed in Table 5.4, both BLP approaches exceeded the available memory and could not be executed. A complete overview of all results, including datasets marked as out-of-memory (OOM), is provided in Section A.1.



(a) Solution quality.



(b) Runtime.



(c) Peak memory consumption.

Figure 5.6: Comparison of CoreColor against BLP_ECC and BLP_CFECC across generated instances. Each subfigure shows a different metric.

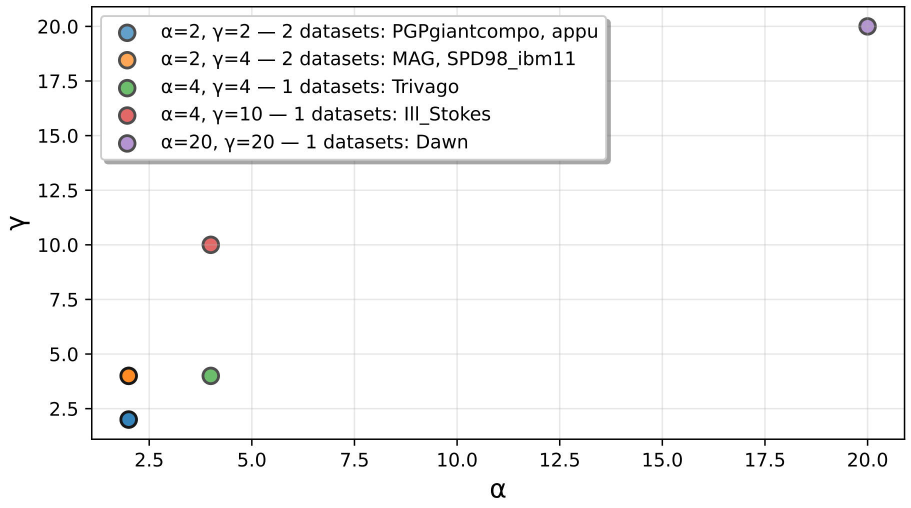


Figure 5.7: Parameter sensitivity for weight factor α together with bonus weights γ.

5.4 CFECC

We now continue with the CFECC problem. As before, we begin with a parameter tuning phase for our FAIRCOLOR algorithm.

Parameter Tuning. To determine suitable parameter values for FAIRCOLOR, we perform a systematic sensitivity analysis on the key parameters: the vote weight factor α together with the bonus weight γ, the number of refinement rounds r, the early-stopping patience p, and the maximum core size $k_{\max}$. All evaluations are carried out on the same set of instances used for CORECOLOR. As β again serves only as a tiebreaker, we keep it fixed without further tuning.

Vote Weight Factor and Worst-Color Bonus. We examine combinations of the vote weight factor α and the bonus weight γ. We set $\alpha \in \{2, 4, 20\}$ and $\gamma \in \{2, 4, 10, 20\}$. All remaining parameters are fixed to $r = 1$, $p = 1$, and $k_{\max} = 55$. Figure 5.7 summarizes the results. While there is no single universally optimal choice, small values of α and γ generally perform well. However, at least one instance achieves its best performance with the largest tested values, $(\alpha, \gamma) = (20, 20)$. To avoid unnecessarily restricting solution quality, we therefore retain the full range of these parameters in our experiments.

Refinement Rounds. Using the previous parameters $\alpha \in \{2, 4, 20\}$ and $\gamma \in \{2, 4, 10, 20\}$, we tune the number of local refinement rounds $r \in \{3, 5, 7, 9, 12\}$, while keeping $p = 1$ and $k_{\max} = 55$ constant.

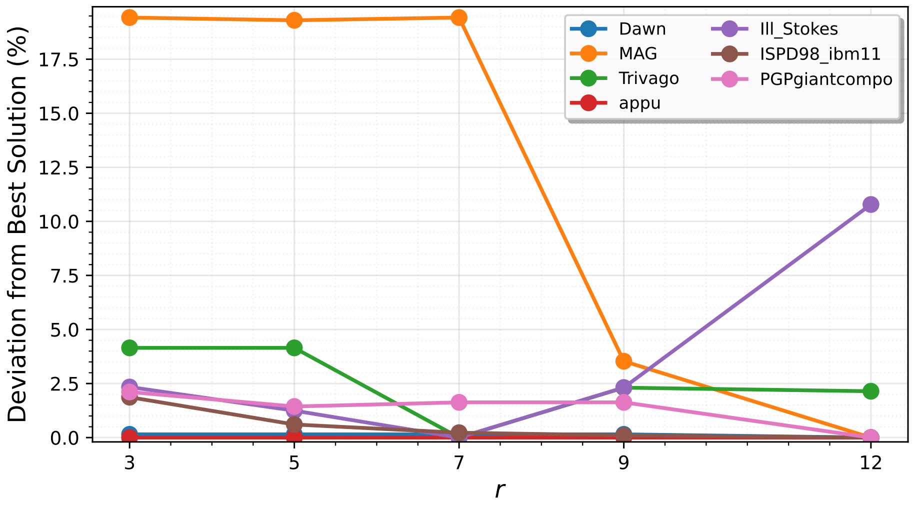


Figure 5.8: Impact of the number of refinement rounds r on solution quality for FAIRCOLOR.

Figure 5.8 shows that the number of refinement rounds has some influence on solution quality. For most graphs, lower values of r already yield good results. However, certain instances, such as MAG, benefit from higher values of r beyond 7. Conversely, very high values can be detrimental: for example, Ill_Stokes experiences a slight decrease in quality when $r = 12$. For the remaining instances, solution quality remains relatively stable across all tested values, with all results within 5% of the best solution. For our final experiments, we set the number of refinement rounds to $r = 9$.

Early Stopping Patience. We also tune the early stopping patience p, similar to our approach for the CORECOLOR algorithm, using values $p \in \{2, 3, 4, 5, 7, 10\}$, while keeping $\alpha \in \{2, 4, 20\}$, $\gamma \in \{2, 4, 10, 20\}$, $r = 9$, and $k_{\max} = 55$ constant.

Figure 5.9 presents the results. We observe that using p values smaller than 5 is generally insufficient. In particular, the MAG dataset shows differences of up to nearly 20%, while most other graphs also benefit from $p > 4$. Therefore we set $p = 5$.

Maximum Core Size. Finally, we tune the parameter $k_{\max}$, which restricts the search space by limiting the number of highest-frequency colors considered as candidates for the core color set. We evaluate $k_{\max}$ in the range from 1 to 55, using the previously selected parameter values $\alpha \in \{2, 4, 20\}$ and $\gamma \in \{2, 4, 10, 20\}$, together with $r = 9$ and $p = 5$.

Figure 5.10 illustrates that $k_{\max}$ indeed influences solution quality. Except for two instances, all graphs benefit from choosing $k_{\max} > 1$, with solution quality improving rapidly for increasing values. However, unlike in the CORECOLOR setting, the improvements stop much earlier. A value of $k_{\max} = 20$ already captures the best or near-best performance across all datasets and will be used.

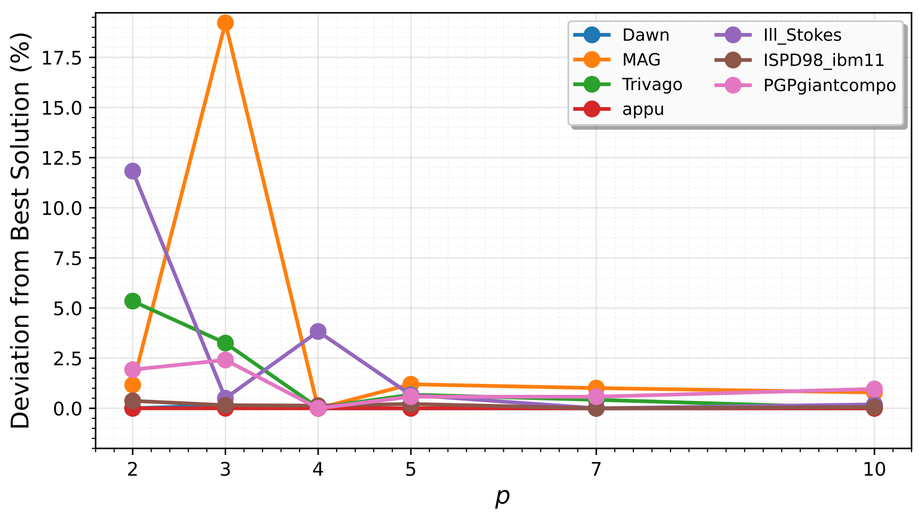


Figure 5.9: Parameter sensitivity for early stopping patience p.

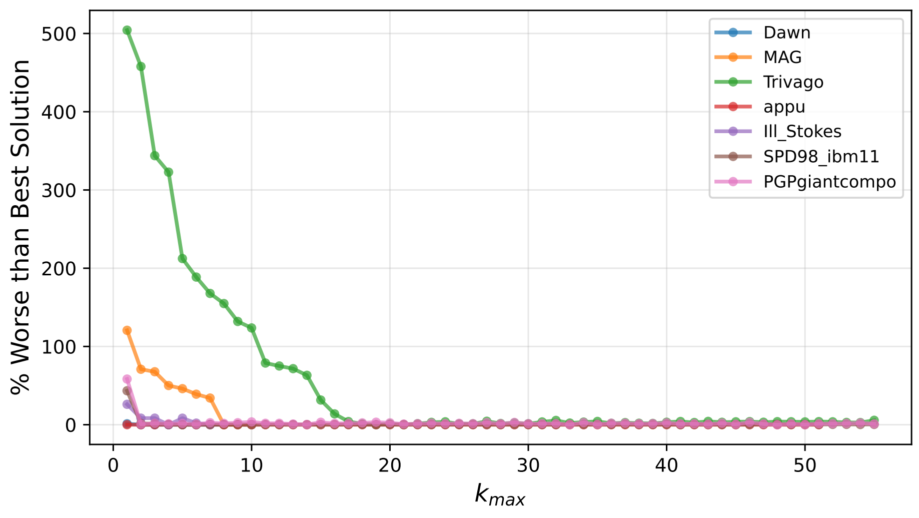


Figure 5.10: Sensitivity of FAIRCOLOR to the maximum core size parameter $k_{\max}$.

After fine-tuning our parameters, we proceed with the final experiments using parameters $\alpha \in \{2, 4, 20\}$ and $\gamma \in \{2, 4, 10, 20\}$, with $r = 9$, $p = 5$, $\beta = 0.01$ and $k_{\max} = 20$. We evaluate FAIRCOLOR against the BLP formulations for MinECC and Color-Fair MinECC proposed by Crane et al. [21]. Our evaluation compares solution quality, runtime, and memory consumption across all benchmark and generated instances. As before, we present the results for benchmark datasets and generated instances separately. We begin with the benchmark datasets.

Solution Quality on Benchmark Sets. Figure 5.11a presents the quality profiles comparing FAIRCOLOR against both BLP formulations. Overall, FAIRCOLOR consistently outperforms BLP_ECC across all instances, while its performance relative to BLP_CFECC varies.

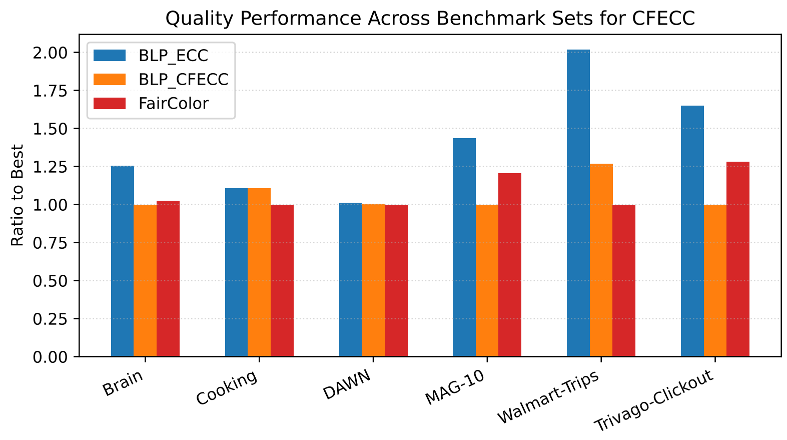
On Brain, the difference is minimal at just 2,3%, and for DAWN, FAIRCOLOR and BLP_CFECC achieve comparable results. In contrast, on MAG-10 and Trivago-Clickout, the BLP formulation slightly outperforms FAIRCOLOR, with relative gaps of 20,6% and 28,3%, respectively. For Walmart-Trips and Cooking, FAIRCOLOR surpasses BLP_CFECC, achieving improvements of 26,9% and 10,7%, respectively. This quality gap reflects the inherent difficulty of balancing fairness across colors using greedy heuristics: while BLP can globally optimize the min-max objective through constraint propagation, the greedy approach makes locally optimal decisions that may disadvantage certain colors early in the construction phase.

Runtime on Benchmark Sets. Figure 5.11b shows that FAIRCOLOR outperforms the BLP_CFECC approach on all instances. For the Walmart-Trips dataset, BLP_CFECC uses a factor of 6,0 more runtime than FAIRCOLOR.

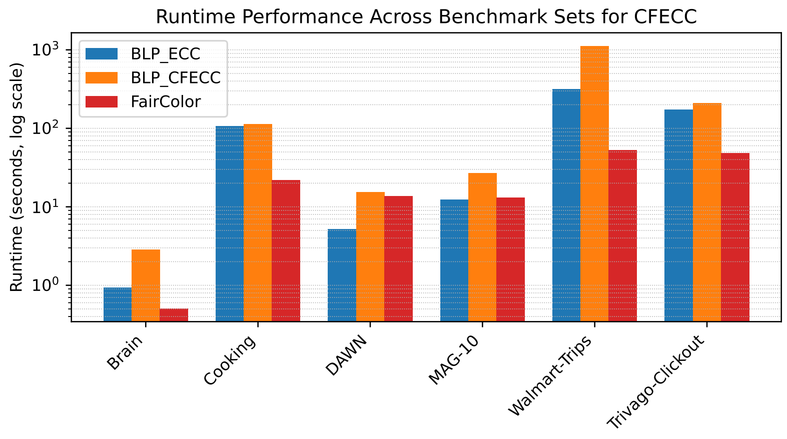
While BLP_CFECC is generally slower, BLP_ECC achieves faster runtimes on some instances. Specifically, FAIRCOLOR requires a factor of 2,7 more time than BLP_ECC for the DAWN dataset, and a factor of 1,1 more time for the MAG-10 dataset.

Memory Consumption on Benchmark Sets. Figure 5.11c presents the memory consumption results, highlighting the advantages of FAIRCOLOR across all instances. Even on the Trivago-Clickout dataset, FAIRCOLOR outperforms both BLP approaches.

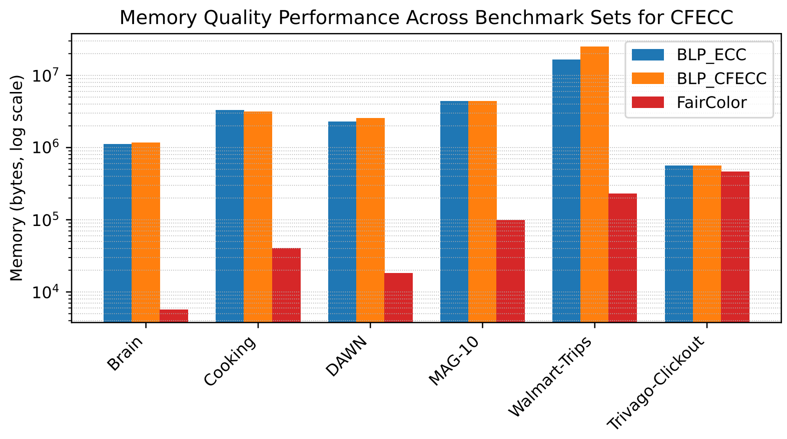
For the Brain dataset, BLP_ECC uses a factor of 194,7 more memory than FAIRCOLOR, while BLP_CFECC consumes 204,2 times more memory. For the Trivago-Clickout dataset, BLP_ECC uses only 1,2 times more memory than FAIRCOLOR, and similarly, BLP_CFECC uses a factor of 1,2 more memory. Interestingly, this is the only dataset where the BLP approaches previously outperformed CORECOLOR.



(a) Solution quality.



(b) Runtime.



(c) Peak memory consumption.

Figure 5.11: Comparison of FairColor against BLP_ECC and BLP_CFECC across benchmark instances. Each subfigure shows a different metric.

We now present the results for the generated instances. We again compare solution quality, runtime, and memory consumption, using performance profiles to do so.

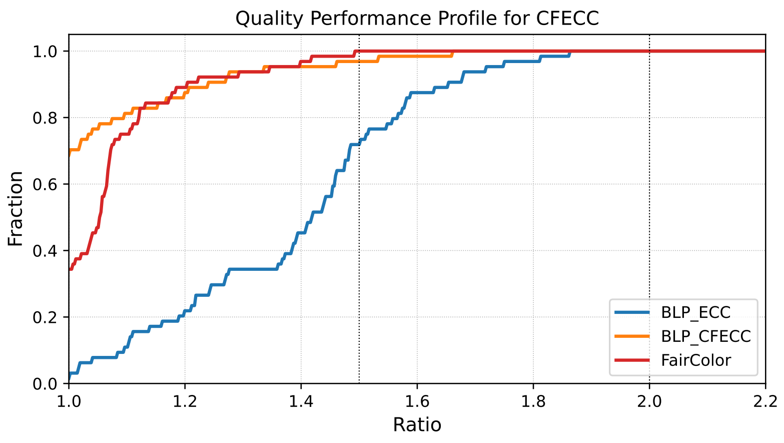
Solution Quality on Generated Sets. Figure 5.11a presents the quality profile comparing FAIRCOLOR against both binary linear programming formulations. In contrast to the results for CORECOLOR, the picture here differs noticeably. FAIRCOLOR achieves the best solution on approximately 34% of the instances, while BLP_CFECC obtains the best result on 68% of them, indicating that the exact BLP formulation for the fairness objective can yield higher-quality solutions.

However, the performance gap narrows substantially as the allowed performance factor increases. Both algorithms converge on roughly 80% of the instances at a factor of 1,12, after which their performance becomes nearly identical. FAIRCOLOR reaches all instances within a worst-case factor of 1,49, whereas BLP_CFECC requires up to 1,66. Moreover, 50% of the BLP_ECC runs lie within a factor of 1,418, with a worst-case value of 1,861.

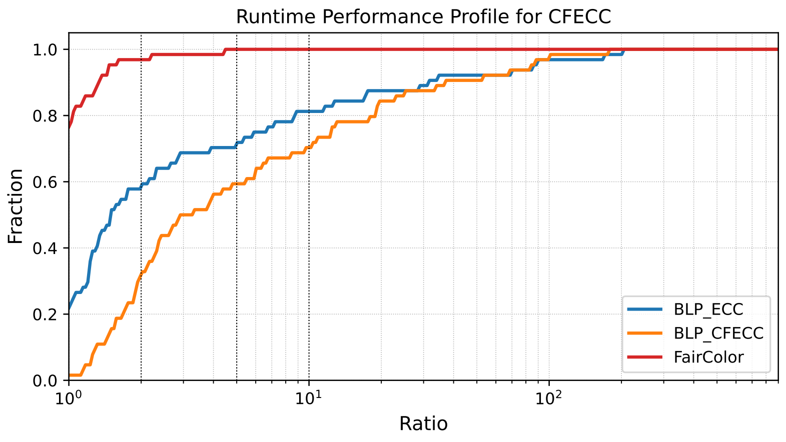
Runtime on Generated Sets. The runtime performance profile in Figure 5.12b shows that FAIRCOLOR behaves similarly to the benchmark sets. It outperforms both BLP formulations in terms of runtime, although the advantage is less pronounced than for CORECOLOR. FAIRCOLOR achieves the fastest runtime on 76,6% of instances, while BLP_ECC is fastest on 21,9% and BLP_CFECC on only 1,5%. As the performance factor increases, BLP_CFECC and BLP_ECC converge, with both reaching approximately 88% of instances at similar factors. In the worst case, BLP_ECC requires up to a factor of 203 longer than FAIRCOLOR, BLP_CFECC up to a factor of 175, whereas FAIRCOLOR's worst-case slowdown is only 4,5. Again, several instances exceed the 30-minute timeout when solved using the BLP formulations. A complete overview of all results, including datasets marked as OOT, is provided in Section A.3.

Memory Consumption on Generated Sets. The memory performance profile in Figure 5.12c closely mirrors the behavior observed for CORECOLOR. FAIRCOLOR consistently outperforms both BLP formulations, requiring substantially less memory across all instances. The two BLP variants exhibit similar memory usage: BLP_ECC shows a median factor of 57,64 and a worst-case factor of 2 658,8, while BLP_CFECC reaches a median of 59,22 and a worst-case factor of 2 786,7. In contrast, FAIRCOLOR's greedy strategy combined with parameter search remains markedly more memory-efficient than either exact BLP formulation.

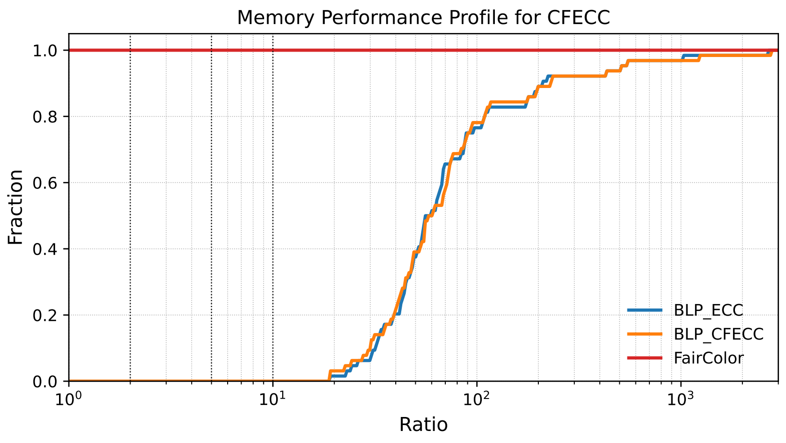
For the large-scale instances listed in Table 5.4, both BLP approaches exceeded the available memory and could not be executed. A complete overview of all results, including datasets that ran out of memory, is provided in Section A.4.



(a) Solution quality.



(b) Runtime.



(c) Peak memory consumption.

Figure 5.12: Comparison of FairColor against BLP_ECC and BLP_CFECC across generated instances. Each subfigure shows a different metric.

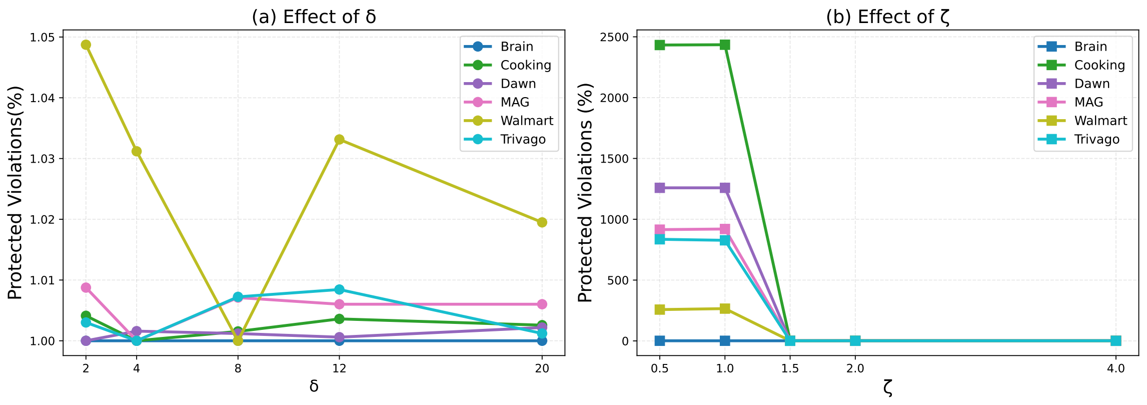


Figure 5.13: Sensitivity of ProtectColor with respect to the parameters δ and ζ.

5.5 PCECC

Finally, we are going over to evalute ProtectColor and again start with a parameter tuning.

Parameter Tuning of δ and ζ. To determine suitable parameter values for ProtectColor, we perform a systematic sensitivity analysis of its key parameters: the preference factor δ, the bonus weight ζ, and the number of refinement rounds r. As β serves again as a tiebreaker, we keep it fixed without further tuning.

We begin by tuning the parameters $\delta \in \{2, 4, 8, 12, 20\}$ and $\zeta \in \{0.5, 1, 1.5, 2, 4\}$ while keeping $r = 1$ fixed. The results are summarized in Figure 5.13. Overall, the parameter δ has little influence on the number of protected color violations, indicating that the algorithm is largely insensitive to δ within the tested range, although for $\delta = 8$ we observe a slight drop in violations for the Walmart dataset. This insensitivity occurs because δ controls the trade-off between vote maximization and conflict minimization during greedy construction, but when protected color constraints dominate, the algorithm's decisions are primarily driven by the need to satisfy those constraints rather than by vote weights. In contrast, ζ shows a more pronounced effect: setting $\zeta = 1.5$ leads to a substantial reduction in violations for five of the six datasets. Increasing ζ beyond 1.5 does not yield further improvements. Based on these observations, we select $\delta = 8$ for the final experiments and $\zeta = 1.5$, which appears to provide the most stable and effective improvement in solution quality.

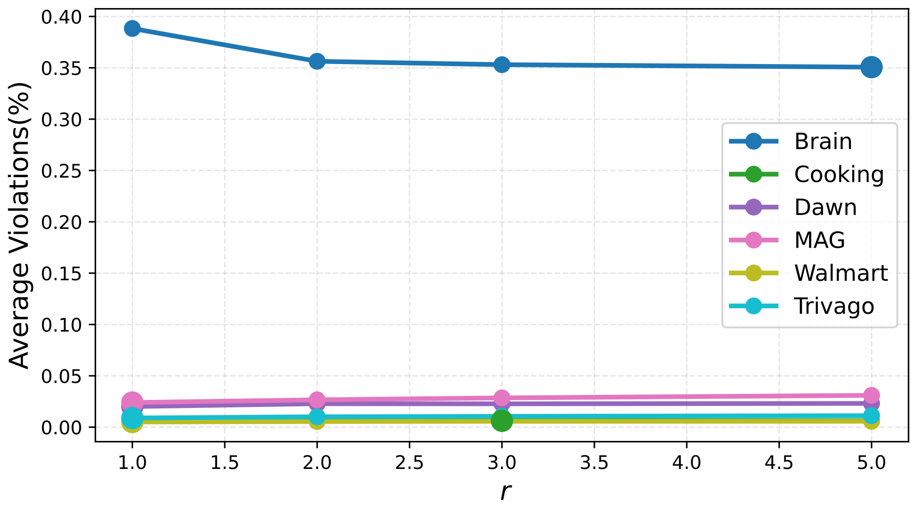


Figure 5.14: Sensitivity of PROTECTCOLOR with respect to refinement rounds r.

Parameter Tuning of r. We now tune the refinement rounds r while keeping $\delta = 8$ and $\zeta = 1.5$. The results are summarized in Figure 5.14. Overall, the number of refinement rounds has only a minor influence on the average protected color violations across our datasets. However, we observe that $r = 2$ provides a small advantage for the Brain dataset. Based on this observation, we select $r = 2$ for all final experiments.

We are now evaluate PROTECTCOLOR by comparing it against the BLP formulation from Crane et al. [21]. We assess two key metrics across varying constraint levels: the number of unsatisfied edges with the protected color (constraint violation) and the overall edge satisfaction across all colors (global quality). For each graph, we protect the median-ranked color by frequency and sweep through constraint levels $\epsilon \in \{0\%, 5\%, 10\%, \dots, 100\%\}$. The results are visualized in Figures 5.15 and 5.15.

For PROTECTCOLOR, we set the following parameters:
weighted vote parameters $\delta = 8$, $\zeta = 1.5$, $\beta = 0.01$, and refinement rounds $r = 2$.

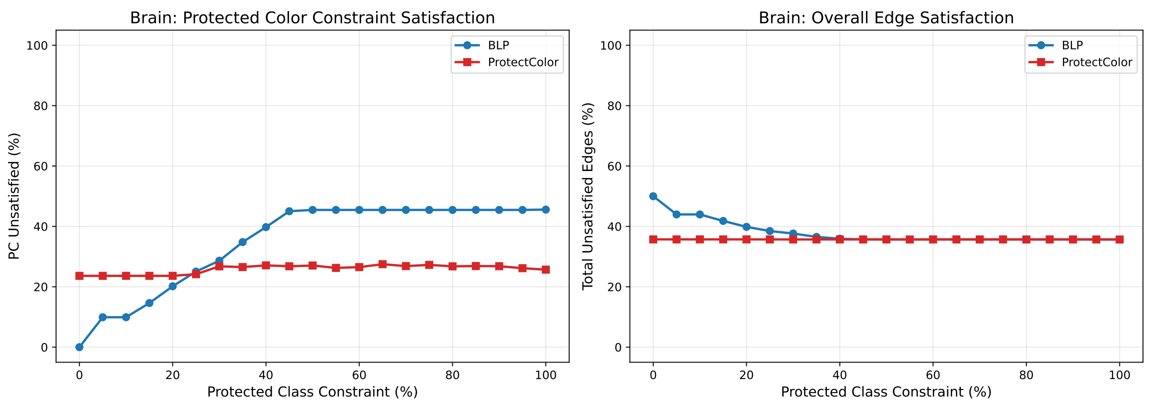
Protected Color Constraint Satisfaction. Figure 5.15 (left columns) shows the number of unsatisfied protected edges as a function of the allowed mistake budget (constraint level). The diagonal line $y = x$ represents the theoretical constraint boundary: points below this line indicate that the algorithm fully satisfies the constraint, while points above violate it. PROTECTCOLOR performs well on most instances, consistently staying at or below the constraint line across all protection levels. On Brain (Figure 5.15a), our PROTECTCOLOR starts slightly above the BLP solution but maintains stable performance, while the BLP deteriorates beyond the 25% constraint level. A notable pattern appears in the 95% to 100% transition: PROTECTCOLOR exhibits a sharp increase in unsatisfied protected edges, which is expected as the 100% level imposes no protection constraint.

The most differences appear on Cooking and Dawn (Figures 5.15b,c). For these instances, the BLP solution experiences dramatic jumps at 45% (Cooking) and 55% (Dawn), sharply increasing the number of unsatisfied protected edges. This is due to the theoretical approximation guarantee of the BLP approach: constraints are guaranteed only up to a factor of 2. When the allowed mistake budget b exceeds roughly 50% of the protected edges, the factor-2 approximation permits leaving many protected edges unsatisfied without violating the guarantee. In contrast, PROTECTCOLOR maintains more consistent protection across all levels.

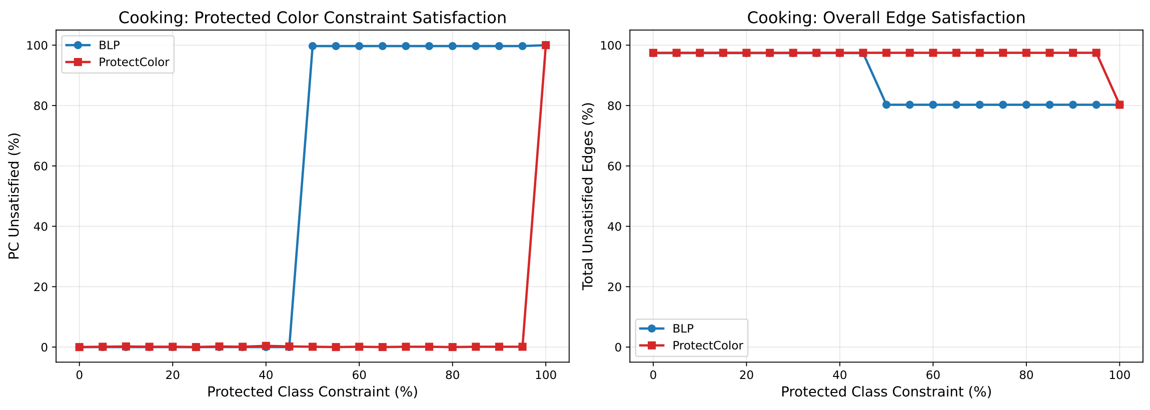
Overall Edge Satisfaction. Figure 5.15 (right columns) show the overall percentage of satisfied edges across all colors as the constraint level varies. For most instances, overall satisfaction remains relatively stable, with a notable drop at 100% for Dawn, Cooking, and MAG (Figures 5.15c, b and 5.15d). At this unconstrained level, PROTECTCOLOR's performance typically converges with BLP, as both optimize for global satisfaction without protection.

The trade-off between protected color satisfaction and global quality is clear on Cooking and Dawn: maintaining better constraint satisfaction comes at the cost of slightly lower overall satisfaction. Conversely, on Brain (Figure 5.15a), PROTECTCOLOR shows slightly worse constraint satisfaction beyond 25% but achieves higher overall edge satisfaction.

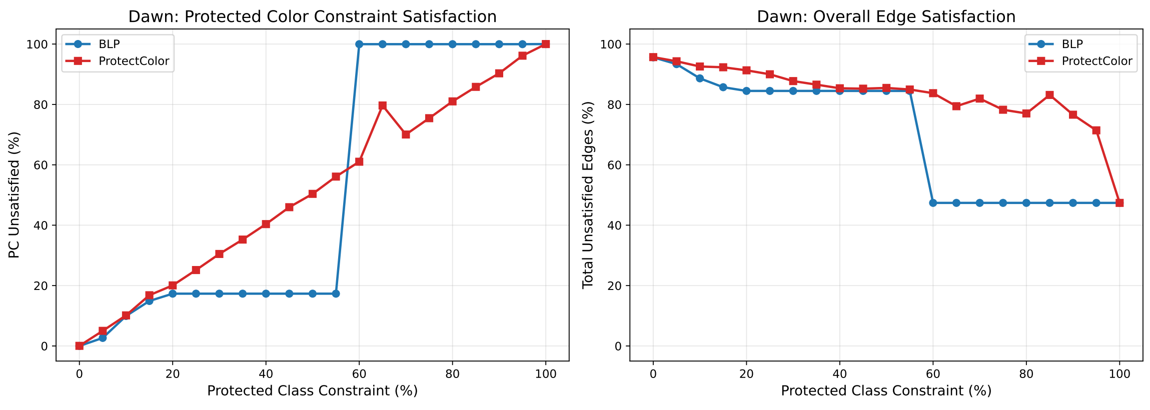
For Walmart, Trivago, and MAG (Figures 5.15e,f and d), differences between PROTECTCOLOR and BLP are minimal, demonstrating that the greedy approach closely approximates the exact optimization on these graphs.



(a) Brain

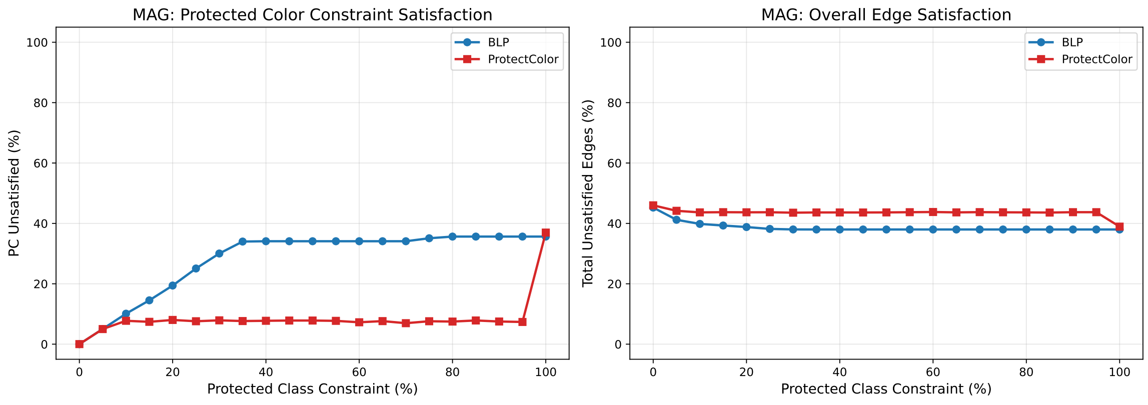


(b) Cooking

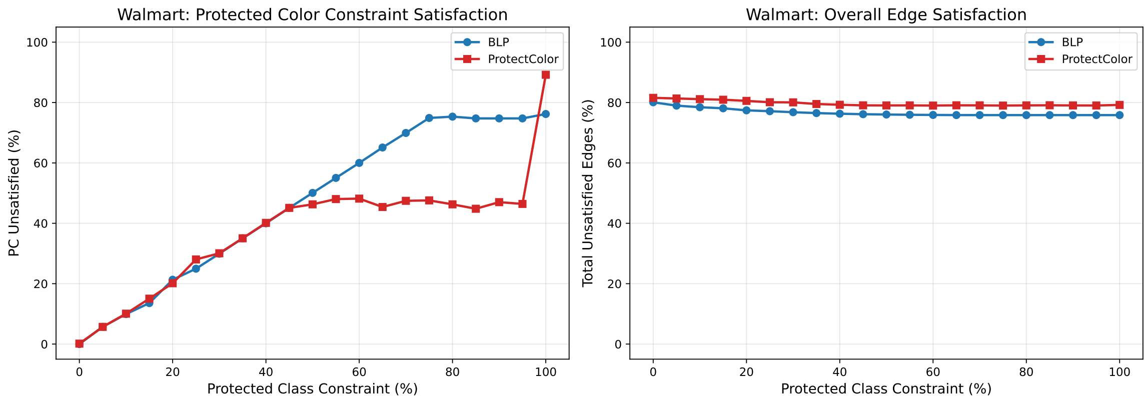


(c) Dawn

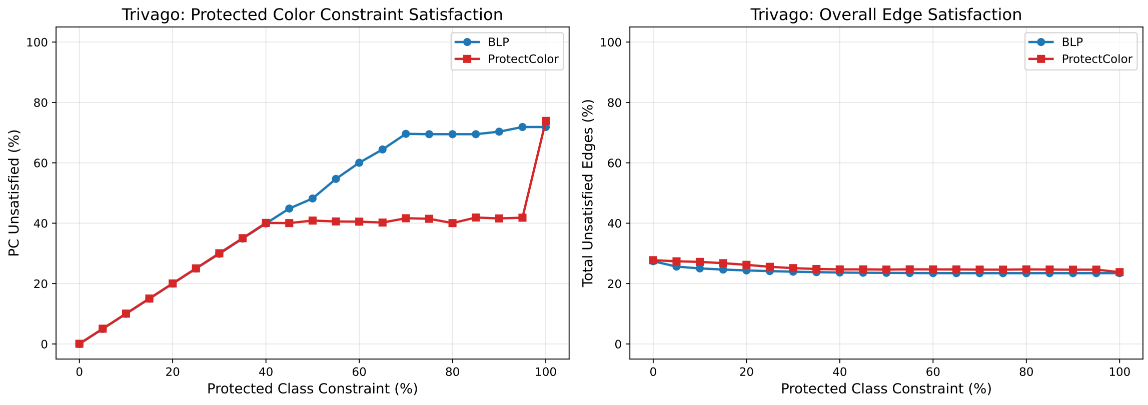
Figure 5.15: Comparison of ProtectColor against the BLP (Part 1)



(d) MAG



(e) Walmart



(f) Trivago

Figure 5.15: Comparison of ProtectColor against the BLP (Part 2, continued)

CHAPTER 6

Discussion

6.1 Conclusion

This thesis introduced and evaluated three greedy algorithms for node coloring in edge-colored hypergraphs, each of them addressing a distinct optimization objective within the ECC framework. The results demonstrate that carefully designed greedy heuristics can achieve favorable trade-offs between solution quality and computational efficiency, both in runtime and memory usage, when compared to exact optimization methods.

CoreColor: Near-Optimal Efficiency. CORECOLOR, our algorithm for the MINECC problem, achieves solution quality nearly indistinguishable from the exact BLP formulation while providing substantial computational advantages. This holds for both the benchmark datasets and the generated instances, where CORECOLOR achieves the fastest runtime on all instances and only underperforms on a single instance in terms of memory consumption. With significant runtime speedups, CORECOLOR demonstrates that the greedy vote-weighted approach with iterative refinement offers an excellent approximation to optimal solutions. These results establish CORECOLOR as highly suitable for large-scale hypergraph clustering, where exact methods become computationally prohibitive.

FairColor: Practical Fairness Optimization. FAIRCOLOR addresses the more challenging CFECC objective through a min-max optimization strategy with parameter grid search. It achieves the best solution on 34% of the generated instances, compared to BLP_CFECC'S 68%. The quality gap, however, narrows substantially across most instances. The same trend holds for the benchmark sets, where BLP_CFECC performs best on three instances, but FAIRCOLOR is close behind and even outperforms the BLP on two instances.

The algorithm maintains significant computational advantages, achieving the fastest runtime on 76,6% of the generated instances and on all benchmark instances, with a worst-case slowdown of only 4,5, compared to BLP'S worst-case factors of 175 to 203. Additionally, FAIRCOLOR outperforms the BLP formulations in terms of memory on every instance. These results demonstrate that greedy heuristics can provide practical fairness-aware clustering when exact optimization is too costly, albeit with a measurable quality trade-off compared to the global optimization achieved by CORECOLOR.

ProtectColor: Empirical Constraint Guarantees. PROTECTCOLOR demonstrates that explicit constraint checking in greedy algorithms can provide tighter practical guarantees than the theoretical worst-case bounds of exact methods. While the BLP formulation offers a factor-2 approximation guarantee that permits significant constraint violations when the mistake budget exceeds 50%, PROTECTCOLOR maintains consistent constraint satisfaction across all protection levels on most instances. This is particularly evident on Cooking and Dawn, where the BLP solution exhibits dramatic jumps in unsatisfied protected edges at the 45% and 55% thresholds, while PROTECTCOLOR maintains stable performance. The algorithm successfully enables systematic exploration of the trade-off between color-specific protection and global clustering quality.

6.2 Future Work

While this thesis proposes three greedy algorithms to solve problems within the ECC framework, there are alternative approaches that could be explored in future work.

Benchmark Expansion. Our evaluation on several instances represents a substantial expansion over prior ECC benchmarks, yet the limited availability of real-world edge-colored hypergraphs remains a bottleneck for comprehensive algorithm evaluation. While synthetically generated instances through b-matching provide valuable insights into algorithm behavior across diverse structural properties, they may not fully capture the complexity and irregularities present in naturally occurring hypergraphs. Future work should prioritize collecting and curating additional real-world edge-colored hypergraphs from diverse application domains such as scientific collaboration networks, biological interaction networks, transportation systems, and e-commerce transaction data. Expanding the benchmark suite beyond the six original real-world instances would enable more robust assessment of algorithm performance and generalizability, ultimately benefiting the broader research community.

Parameter Optimization and Reduction. Our algorithms currently require manual specification of several parameters. While these parameters enable flexible algorithm behavior and have been empirically validated, their presence complicates practical deployment and may limit accessibility for practitioners unfamiliar with the algorithms' internal workings. Future research could pursue two complementary directions.

First, investigating *parameter-free* or *self-tuning* variants that automatically adapt parameter values based on graph structure could help reduce manual tuning effort. For instance, the refinement rounds r might be determined dynamically by monitoring convergence rates, or the patience parameter p could adapt based on the observed improvement trajectory. Similarly, the vote weight factor α might be inferred from local graph properties such as edge density or color distribution.

Second, running systematic tests where we vary one parameter at a time could show which parameters really matter for performance and which ones can safely be fixed to stable default values. For FAIRCOLOR specifically, analyzing whether the grid search over $\alpha \times \gamma$ combinations could be replaced with a single adaptive parameter or a coarser search space may reduce computational overhead while maintaining solution quality. Such simplifications would make the algorithms more practical for deployment while likely incurring only modest increases in runtime or memory consumption.

Alternative Optimization Approaches. This thesis focused exclusively on greedy heuristics for ECC. Future research could explore alternative algorithmic paradigms that may offer different trade-offs between solution quality and computational efficiency.

Hybrid methods represent one promising direction. Combining exact and heuristic approaches could leverage the strengths of both. For instance, using BLP formulations to solve small subproblems or critical graph regions, followed by greedy refinement on the complete solution, might improve quality while maintaining computational tractability. Alternatively, warm-starting BLP solvers with greedy solutions could accelerate convergence by providing high-quality initial feasible solutions.

Another promising direction is to explore spectral clustering techniques. By constructing appropriate Laplacian matrices that encode both hypergraph structure and edge colors, spectral methods could identify natural color assignments through eigenvector analysis. Such approaches have proven effective for traditional graph clustering and could be adapted to the edge-colored setting, potentially offering both theoretical guarantees and practical efficiency.

Finally, with the rise of neural networks, one could try to develop a graph neural network that could help predict high-quality node colorings directly from hypergraph structures. Training such networks on instances with known optimal solutions might yield models that generalize to unseen graphs, potentially providing both speed and quality benefits.

APPENDIX A

Appendix

A.1 Results MinECC

The following tables lists all instances along with their runtime and memory consumption for MinECC.

Table A.1: Comparison of runtime in seconds for BLP_ECC, BLP_CFECC, and CoreColor.

Instance	BLP_ECC	BLP_CFECC	CoreColor
Brain	0,941	2,853	0,047
Cooking	106,431	113,340	1,167
DAWN	5,175	15,313	0,598
MAG-10	12,301	26,728	1,176
Walmart-Trips	317,077	1 122,516	7,876
Trivago-Clickout	172,793	208,976	10,076
192bit	297,338	307,384	0,216
ABACUS_shell_hd	OOT	OOT	0.230
airfoil_2d	OOT	OOT	0.210
Amazon	OOT	OOT	1 794,70
Andrews	21,314	791,828	1,001
appu	558,783	350,021	10,070
as-22july06	77,355	66,885	0,215
as-caida	96,627	93,767	0,248
astro-ph	27,990	23,040	0,648
av41092	OOT	OOT	2.842
bayer04	7,040	7,945	0,186

Continued on next page

Instance	**BLP_ECC**	**BLP_CFECC**	**CoreColor**
bcsstk29	117,258	12,847	0,729
deltaX	101,814	208,348	0,385
epb1	1,941	28,607	0,072
EternityII_A	12,566	29,264	1,511
ex19	10,464	27,775	0,399
fd18	0,913	3,443	0,048
finan512	48,069	67,972	1,059
foldoc	28,860	51,939	0,323
Franz11	1 505,501	737,139	0,290
G2_circuit	17,060	93,056	0,581
G67	0,374	1,501	0,027
g7jac040sc	5,273	9,350	0,202
garon2	71,800	128,565	0,426
gemat1	1,135	1,484	0,256
graphics	6,691	81,927	0,263
gyro	45,060	50,354	1,486
hvdc1	3,954	5,640	0,170
IG5-17	110,909	732,345	2,655
Ill_Stokes	385,258	292,000	0,169
image_interp	15,889	41,421	0,471
ISPD98_ibm01	2,341	3,493	0,074
ISPD98_ibm02	5,951	10,093	0,142
ISPD98_ibm03	9,458	9,206	0,229
ISPD98_ibm04	64,864	61,653	0,356
ISPD98_ibm05	5,304	19,335	0,137
ISPD98_ibm06	12,287	18,789	0,277
ISPD98_ibm07	19,119	25,117	0,365
ISPD98_ibm08	280,645	319,181	0,651
ISPD98_ibm09	44,245	47,651	0,768
ISPD98_ibm10	43,321	60,426	1,003
ISPD98_ibm11	58,128	64,797	0,992
ISPD98_ibm12	164,804	183,478	1,129
ISPD98_ibm13	74,703	90,081	1,422
ISPD98_ibm14	208,100	226,583	2,293
ISPD98_ibm15	250,955	318,994	3,621
ISPD98_ibm16	180,715	246,961	3,862
ISPD98_ibm17	109,175	216,075	2,564
ISPD98_ibm18	112,244	209,017	2,537

Continued on next page

Instance	**BLP_ECC**	**BLP_CFECC**	**CoreColor**
language	178,559	240,015	5,320
light_in_tissue	13,585	772,934	0,330
lp_nug20	51,594	52,818	0,422
lhr14	OOT	OOT	0.262
lung2	13,238	16,861	0,397
Maggeo	OOT	OOT	192.550
Maragal_6	36,707	36,648	0,708
mri1	133,641	167,009	1,777
msc10848	90,151	63,379	1,514
nopoly	1,284	12,294	0,056
NotreDame_actors	156,266	179,414	6,530
obstclae	3,328	17,193	0,119
opt1	793,308	100,396	2,698
Oregon-1	88,392	86,624	0,093
p2p-Gnutella25	7,205	5,612	0,088
Pd_rhs	0,305	0,325	0,018
pesa	1,508	28,741	0,061
PGPgiantcompo	4,752	4,998	0,152
poli3	2,205	3,031	0,068
powersim	2,028	2,160	0,062
psse2	3,917	3,795	0,215
StackOverflow	OOT	OOT	1 260,70

Table A.2: Comparison of memory in kilobytes for BLP_ECC, BLP_CFECC, and CORECOLOR.

Instance	**BLP_ECC**	**BLP_CFECC**	**CoreColor**
Brain	1 116 148	1 170 460	5 468
Cooking	3 294 800	3 146 496	40 220
DAWN	2 302 832	2 544 408	18 040
MAG-10	4 438 144	4 414 088	90 416
Walmart-Trips	16 528 708	24 892 972	550 392
Trivago-Clickout	564 260	566 160	1 066 632
192bit	85 464 532	89 574 328	57 276
ABACUS_shell_hd	-	-	42 592
airfoil_2d	-	-	20 200
Amazon	OOM	OOM	15 949 444
Andrews	6 416 676	6 857 228	89 800

Continued on next page

Table A.2 – continued from previous page

Instance	**BLP_ECC**	**BLP_CFECC**	**CoreColor**
appu	12 037 756	10 501 636	869 760
as-22july06	23 915 972	23 930 600	84 452
as-caida	32 541 964	32 349 056	95 140
astro-ph	7 565 252	8 433 080	239 656
av41092	-	-	755 020
bayer04	2 313 852	2 480 168	68 512
bcsstk29	4 590 456	4 206 892	268 964
deltaX	4 437 448	4 467 288	93 556
epb1	1 324 188	1 458 728	18 772
EternityII_A	3 642 492	4 146 212	65 844
ex19	2 331 972	2 293 792	120 076
fd18	1 120 448	1 150 012	12 020
finan512	8 714 808	8 733 500	324 296
foldoc	12 464 496	11 508 096	133 900
Franz11	17 012 172	3 661 844	91 840
G2_circuit	5 618 908	6 202 160	96 024
G67	891 968	957 396	8 040
g7jac040sc	2 478 968	2 538 312	92 536
garon2	2 633 236	2 626 800	132 536
gemat1	1 088 236	1 104 784	21 776
graphics	2 077 952	2 239 564	87 484
gyro	9 053 368	8 779 716	675 288
hvdc1	1 950 932	2 071 432	73 392
IG5-17	25 787 992	25 676 516	453 900
Ill_Stokes	2 668 472	2 948 448	39 092
image_interp	5 545 180	6 088 692	108 676
ISPD98_ibm01	1 461 896	1 506 956	28 464
ISPD98_ibm02	2 714 796	2 647 876	70 940
ISPD98_ibm03	3 335 540	18 010 328	125 228
ISPD98_ibm04	17 025 136	1 690 436	160 464
ISPD98_ibm05	1 690 436	2 318 536	28 848
ISPD98_ibm06	4 645 160	4 653 664	155 032
ISPD98_ibm07	7 201 980	7 215 772	225 452
ISPD98_ibm08	70 802 516	71 205 152	282 520
ISPD98_ibm09	12 545 684	12 574 788	357 552
ISPD98_ibm10	12 821 748	13 108 044	465 680
ISPD98_ibm11	15 572 368	16 381 052	451 196
ISPD98_ibm12	43 954 560	43 939 920	470 272

Continued on next page

Table A.2 – continued from previous page

Instance	BLP_ECC	BLP_CFECC	CoreColor
ISPD98_ibm13	20 158 032	19 962 280	609 580
ISPD98_ibm14	51 807 232	51 742 120	1 015 664
ISPD98_ibm15	62 338 884	62 792 536	1 260 152
ISPD98_ibm16	39 321 920	44 920 552	1 329 144
ISPD98_ibm17	22 256 836	19 892 692	868 872
ISPD98_ibm18	28 340 940	28 208 492	1 049 904
language	58 065 544	61 898 596	2 289 676
light_in_tissue	2 601 004	3 316 584	70 160
lp_nug20	5 850 988	5 874 420	171 496
lhr14	-	-	86080
lung2	3 651 400	3 540 324	83 324
Maggeo	OOM	OOM	7981180
Maragal_6	10 523 016	10 803 592	218 456
mri1	36 593 152	36 598 184	926 400
msc10848	10 180 344	10 161 088	525 104
nopoly	1 145 120	1 154 892	12 344
NotreDame_actors	45 315 152	46 686 936	2 450 800
obstclae	1 999 776	2 180 392	28 060
opt1	10 196 424	9 175 344	907 876
Oregon-1	25 944 372	31 557 908	43 620
p2p-Gnutella25	1 654 796	1 688 604	38 680
Pd_rhs	878 652	891 684	12 176
pesa	1 219 124	1 307 040	13 176
PGPgiantcompo	3 531 144	3 215 744	75 940
poli3	1 583 168	1 513 740	32 276
powersim	1 510 700	1 549 048	37 364
psse2	1 855 000	1 885 952	74 604
StackOverflow	OOM	OOM	17 575 716

A.2 Results CFECC

The following table lists all instances along with their runtime and memory consumption for CFECC.

Table A.3: Comparison of runtime in seconds for BLP_ECC, BLP_CFECC, and CORECOLOR.

Instance	BLP_ECC	BLP_CFECC	FairColor
Brain	0,941	2,853	0,047
Cooking	106,431	113,340	1,167
DAWN	5,175	15,313	0,598
MAG-10	12,301	26,728	1,176
Walmart-Trips	317,077	1 122,516	7,876
Trivago-Clickout	172,793	208,976	10,076
192bit	297,338	307,384	0,216
ABACUS_shell_hd	OOT	OOT	13,396
airfoil_2d	OOT	OOT	3,810
Amazon	OOT	OOT	2 154
Andrews	21,314	791,828	1,001
appu	558,783	350,021	10,070
as-22july06	77,355	66,885	0,215
as-caida	96,627	93,767	0,248
astro-ph	27,990	23,040	0,648
av41092	OOT	OOT	35,841
bayer04	7,040	7,945	0,186
bcsstk29	117,258	12,847	0,729
deltaX	101,814	208,348	0,385
epb1	1,941	28,607	0,072
EternityII_A	12,566	29,264	1,511
ex19	10,464	27,775	0,399
fd18	0,913	3,443	0,048
finan512	48,069	67,972	1,059
foldoc	28,860	51,939	0,323
Franz11	1 505,501	737,139	0,290
G2_circuit	17,060	93,056	0,581
G67	0,374	1,501	0,027
g7jac040sc	5,273	9,350	0,202
garon2	71,800	128,565	0,426
gemat1	1,135	1,484	0,256
graphics	6,691	81,927	0,263

Continued on next page

Instance	**BLP_ECC**	**BLP_CFECC**	**FairColor**
gyro	45,060	50,354	1,486
hvdc1	3,954	5,640	0,170
IG5-17	110,909	732,345	2,655
Ill_Stokes	385,258	292,000	0,169
image_interp	15,889	41,421	0,471
ISPD98_ibm01	2,341	3,493	0,074
ISPD98_ibm02	5,951	10,093	0,142
ISPD98_ibm03	9,458	9,206	0,229
ISPD98_ibm04	64,864	61,653	0,356
ISPD98_ibm05	5,304	19,335	0,137
ISPD98_ibm06	12,287	18,789	0,277
ISPD98_ibm07	19,119	25,117	0,365
ISPD98_ibm08	280,645	319,181	0,651
ISPD98_ibm09	44,245	47,651	0,768
ISPD98_ibm10	43,321	60,426	1,003
ISPD98_ibm11	58,128	64,797	0,992
ISPD98_ibm12	164,804	183,478	1,129
ISPD98_ibm13	74,703	90,081	1,422
ISPD98_ibm14	208,100	226,583	2,293
ISPD98_ibm15	250,955	318,994	3,621
ISPD98_ibm16	180,715	246,961	3,862
ISPD98_ibm17	109,175	216,075	2,564
ISPD98_ibm18	112,244	209,017	2,537
language	178,559	240,015	5,320
light_in_tissue	13,585	772,934	0,330
lp_nug20	51,594	52,818	0,422
lhr14	OOT	OOT	2,175
lung2	13,238	16,861	0,397
Maggeo	OOT	OOT	92,290
Maragal_6	36,707	36,648	0,708
mri1	133,641	167,009	1,777
msc10848	90,151	63,379	1,514
nopoly	1,284	12,294	0,056
NotreDame_actors	156,266	179,414	6,530
obstclae	3,328	17,193	0,119
opt1	793,308	100,396	2,698
Oregon-1	88,392	86,624	0,093
p2p-Gnutella25	7,205	5,612	0,088

Continued on next page

Instance	BLP_ECC	BLP_CFECC	FairColor
Pd_rhs	0,305	0,325	0,018
pesa	1,508	28,741	0,061
PGPgiantcompo	4,752	4,998	0,152
poli3	2,205	3,031	0,068
powersim	2,028	2,160	0,062
psse2	3,917	3,795	0,215
StackOverflow	OOT	OOT	369.199

Table A.4: Comparison of memory in kilobytes for BLP_ECC, BLP_CFECC, and FairColor.

Instance	BLP_ECC	BLP_CFECC	FairColor
Brain	1 116 148	1 170 460	5 468
Cooking	3 294 800	3 146 496	40 220
DAWN	2 302 832	2 544 408	18 040
MAG-10	4 438 144	4 414 088	90 416
Walmart-Trips	16 528 708	24 892 972	550 392
Trivago-Clickout	564 260	566 160	1 066 632
192bit	85 464 532	89 574 328	57 276
ABACUS_shell_hd	-	-	33420
airfoil_2d	-	-	20948
Amazon	OOM	OOM	3 289 144
Andrews	6 416 676	6 857 228	89 800
appu	12 037 756	10 501 636	869 760
as-22july06	23 915 972	23 930 600	84 452
as-caida	32 541 964	32 349 056	95 140
astro-ph	7 565 252	8 433 080	239 656
av41092	-	-	74 764
bayer04	2 313 852	2 480 168	68 512
bcsstk29	4 590 456	4 206 892	268 964
deltaX	4 437 448	4 467 288	93 556
epb1	1 324 188	1 458 728	18 772
EternityII_A	3 642 492	4 146 212	65 844
ex19	2 331 972	2 293 792	120 076
fd18	1 120 448	1 150 012	12 020
finan512	8 714 808	8 733 500	324 296
foldoc	12 464 496	11 508 096	133 900
Franz11	17 012 172	3 661 844	91 840

Continued on next page

Table A.4 – continued from previous page

Instance	BLP_ECC	BLP_CFECC	FairColor
G2_circuit	5 618 908	6 202 160	96 024
G67	891 968	957 396	8 040
g7jac040sc	2 478 968	2 538 312	92 536
garon2	2 633 236	2 626 800	132 536
gemat1	1 088 236	1 104 784	21 776
graphics	2 077 952	2 239 564	87 484
gyro	9 053 368	8 779 716	675 288
hvdc1	1 950 932	2 071 432	73 392
IG5-17	25 787 992	25 676 516	453 900
Ill_Stokes	2 668 472	2 948 448	39 092
image_interp	5 545 180	6 088 692	108 676
ISPD98_ibm01	1 461 896	1 506 956	28 464
ISPD98_ibm02	2 714 796	2 647 876	70 940
ISPD98_ibm03	3 335 540	18 010 328	125 228
ISPD98_ibm04	17 025 136	1 690 436	160 464
ISPD98_ibm05	1 690 436	2 318 536	28 848
ISPD98_ibm06	4 645 160	4 653 664	155 032
ISPD98_ibm07	7 201 980	7 215 772	225 452
ISPD98_ibm08	70 802 516	71 205 152	282 520
ISPD98_ibm09	12 545 684	12 574 788	357 552
ISPD98_ibm10	12 821 748	13 108 044	465 680
ISPD98_ibm11	15 572 368	16 381 052	451 196
ISPD98_ibm12	43 954 560	43 939 920	470 272
ISPD98_ibm13	20 158 032	19 962 280	609 580
ISPD98_ibm14	51 807 232	51 742 120	1 015 664
ISPD98_ibm15	62 338 884	62 792 536	1 260 152
ISPD98_ibm16	39 321 920	44 920 552	1 329 144
ISPD98_ibm17	22 256 836	19 892 692	868 872
ISPD98_ibm18	28 340 940	28 208 492	1 049 904
language	58 065 544	61 898 596	2 289 676
light_in_tissue	2 601 004	3 316 584	70 160
lp_nug20	5 850 988	5 874 420	171 496
lhr14	-	-	23004
lung2	3 651 400	3 540 324	83 324
Maggeo	OOM	OOM	725576
Maragal_6	10 523 016	10 803 592	218 456
mri1	36 593 152	36 598 184	926 400
msc10848	10 180 344	10 161 088	525 104

Continued on next page

Table A.4 – continued from previous page

Instance	**BLP_ECC**	**BLP_CFECC**	**FairColor**
nopoly	1 145 120	1 154 892	12 344
NotreDame_actors	45 315 152	46 686 936	2 450 800
obstclae	1 999 776	2 180 392	28 060
opt1	10 196 424	9 175 344	907 876
Oregon-1	25 944 372	31 557 908	43 620
p2p-Gnutella25	1 654 796	1 688 604	38 680
Pd_rhs	878 652	891 684	12 176
pesa	1 219 124	1 307 040	13 176
PGPgiantcompo	3 531 144	3 215 744	75 940
poli3	1 583 168	1 513 740	32 276
powersim	1 510 700	1 549 048	37 364
psse2	1 855 000	1 885 952	74 604
StackOverflow	OOM	OOM	2403984

Zusammenfassung

Edge-Colored Clustering (ECC) ist ein Hypergraph-Partitionierungsproblem, bei dem Kanten mit Farben gefärbt sind und das Ziel darin besteht, Knoten so einzufärben, dass sie die gleiche Farbe wie die Kante haben. Das Framework findet Anwendung in Community Detection, Empfehlungssystemen und Data Mining, wo heterogene Beziehungen die Clustering-Struktur beeinflussen. Obwohl aktuelle Arbeiten theoretische Grundlagen und exakte Optimierungsverfahren bieten, bleiben skalierbare Algorithmen für große Hypergraphen begrenzt.

Diese Arbeit entwickelt und evaluiert drei Greedy-Algorithmen für die Knotenfärbung in edge-colored Hypergraphen, die jeweils auf unterschiedliche Optimierungsziele zugeschnitten sind. Wir führen eine Core-Color-Priorisierungsstrategie ein, die die Farbpalette auf die k häufigsten Kantenfarben beschränkt und dadurch die Skalierbarkeit verbessert, indem sie sich auf dominante Strukturmuster konzentriert. Die resultierenden Algorithmen – CoreColor, FairColor und ProtectColor – adressieren die Minimierung global unerfüllter Kanten, die Erfüllung über Farben hinweg und constraint-basierte Optimierung mit geschützten Farben.

CoreColor verwendet vote-weighted Greedy-Initialisierung mit lokaler Verfeinerung, um die Gesamtzahl unerfüllter Kanten zu minimieren. FairColor wendet ein Min-Max-Fairness-Ziel mittels Parameter-Grid-Search an, und ProtectColor erzwingt minimale Erfüllungsgarantien für festgelegte geschützte Farben bei gleichzeitiger Optimierung der globalen Qualität.

Experimente auf diversen Hypergraph-Instanzen zeigen, dass unsere Greedy-Algorithmen eine Lösungsqualität nahe exakter Binary Linear Programming (BLP)-Methoden erreichen und dabei erhebliche Rechenvorteile bieten. CoreColor erreicht BLP-Niveau-Qualität, ist jedoch bis zu 5 191-mal schneller und bis zu 1 563-mal speichereffizienter. FairColor liefert konkurrenzfähige fairness-bewusste Lösungen mit einer verbesserten Laufzeit von bis zu einem Faktor von 203 und Speicherreduktionen bis zu 2 786. ProtectColor zeigt darüber hinaus, dass constraint-erzwungene Greedy-Methoden bessere praktische Garantien liefern können als die theoretischen Worst-Case-Schranken exakter Ansätze.

Insgesamt zeigen die Ergebnisse, dass sorgfältige Greedy-Heuristiken eine exzellente Balance zwischen Lösungsqualität und Recheneffizienz bieten und es ermöglichen, ECC auf Instanzen zu skalieren, die jenseits der Reichweite exakter Methoden liegen.

Bibliography

[1] Alexander Ageev and Kononov Alexander. Improved approximations for the max k-colored clustering problem. pages 1–10, 09 2014.

[2] Alexander Ageev and Kononov Alexander. *A 0.3622-Approximation Algorithm for the Maximum k-Edge-Colored Clustering Problem*, pages 3–15. 09 2020.

[3] Yousef M. Alhamdan and Alexander Kononov. Approximability and inapproximability for maximum k-edge-colored clustering problem. In *Computer Science - Theory and Applications: 14th International Computer Science Symposium in Russia, CSR 2019, Novosibirsk, Russia, July 1-5, 2019, Proceedings*, pages 1–12, Berlin, Heidelberg, 2019. Springer-Verlag.

[4] Charles J. Alpert. The ispd98 circuit benchmark suite. In *Proceedings of the 1998 International Symposium on Physical Design*, ISPD '98, pages 80–85, New York, NY, USA, 1998. Association for Computing Machinery.

[5] Ilya Amburg, Nate Veldt, and Austin R. Benson. *Diverse and Experienced Group Discovery via Hypergraph Clustering*, pages 145–153.

[6] Ilya Amburg, Nate Veldt, and Austin R. Benson. Hypergraph clustering with categorical edge labels. *CoRR*, abs/1910.09943, 2019.

[7] Yael Anava, Noa Avigdor-Elgrabli, and Iftah Gamzu. Improved theoretical and practical guarantees for chromatic correlation clustering. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, pages 55–65, Republic and Canton of Geneva, CHE, 2015. International World Wide Web Conferences Steering Committee.

[8] Eric Angel, Evripidis Bampis, Kononov Alexander, Dimitris Paparas, Emmanouil Pountourakis, and Vassilis Zissimopoulos. Clustering on k-edge-colored graphs. 08 2013.

[9] Nikhil Bansal, Avrim Blum, and Shuchi Chawla. Correlation clustering. *Machine Learning*, 56(1):89–113, 2004.

[10] Austin R. Benson. Three hypergraph eigenvector centralities. *SIAM Journal on Mathematics of Data Science*, 1(2):293–312, 2019.

[11] Francesco Bonchi, Aristides Gionis, Francesco Gullo, Charalampos E. Tsourakakis, and Antti Ukkonen. Chromatic correlation clustering. *ACM Trans. Knowl. Discov. Data*, 9(4), June 2015.

[12] Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Antti Ukkonen. Chromatic correlation clustering. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pages 1321–1329, New York, NY, USA, 2012. Association for Computing Machinery.

[13] Leizhen CAI and On Yin LEUNG. Alternating path and coloured clustering, 2018.

[14] Gruia Calinescu, Howard Karloff, and Yuval Rabani. An improved approximation algorithm for multiway cut. *J. Comput. Syst. Sci.*, 60(3):564–574, June 2000.

[15] Simon Caton and Christian Haas. Fairness in machine learning: A survey. *ACM Computing Surveys*, 56(7):1–38, April 2024.

[16] Chandra Chekuri and Alina Ene. Approximation algorithms for submodular multiway partition, 2011.

[17] Chandra Chekuri and Alina Ene. Submodular cost allocation problem and applications, 2011.

[18] Chandra Chekuri and Vivek Madan. *Simple and Fast Rounding Algorithms for Directed and Node-weighted Multiway Cut*, pages 797–807.

[19] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, and Sergei Vassilvitskii. Fair clustering through fairlets, 2018.

[20] Alex Crane, Brian Lavallee, Blair D. Sullivan, and Nate Veldt. Overlapping and robust edge-colored clustering in hypergraphs, 2024.

[21] Alex Crane, Thomas Stanley, Blair D. Sullivan, and Nate Veldt. Edge-colored clustering in hypergraphs: Beyond minimizing unsatisfied edges, 2025.

[22] Nicolas Crossley, Andrea Mechelli, Petra Vértes, Toby Winton-Brown, Ameera Patel, Cedric Ginestet, Philip Mcguire, and Edward Bullmore. Cognitive relevance of the community structure of the human brain functional coactivation network. *Proceedings of the National Academy of Sciences of the United States of America*, 110, 06 2013.

[23] Elias Dahlhaus, David Johnson, Christos Papadimitriou, Paul Seymour, and Mihalis Yannakakis. The complexity of multiterminal cuts. *SIAM J. Comput.*, 23:864–894, 08 1994.

[24] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), December 2011.

[25] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, 2002.

[26] Alina Ene, Jan Vondrak, and Yi Wu. Local distribution and the symmetry gap: Approximability of multiway partitioning problems, 2015.

[27] D.R. Forsyth. *Group Dynamics*. Cengage Learning, 2018.

[28] Kimon Fountoulakis, Pan Li, and Shenghao Yang. Local hyper-flow diffusion. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 27683–27694. Curran Associates, Inc., 2021.

[29] Naveen Garg, Vijay V. Vazirani, and Mihalis Yannakakis. Multiway cuts in node weighted graphs. *Journal of Algorithms*, 50(1):49–61, 2004.

[30] Ernestine Großmann, Felix Joos, Henrik Reinstädtler, and Christian Schulz. Engineering hypergraph b-matching algorithms, 2024.

[31] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2024.

[32] Matthias Hein, Simon Setzer, Leonardo Jost, and Syama Sundar Rangapuram. The total variation on hypergraphs - learning on hypergraphs revisited, 2013.

[33] Susan Jackson and Marian Ruderman. Diversity in work teams: Research paradigms for a changing workplace. *Diversity in work teams*, 01 1995.

[34] Leon Kellerhals, Tomohiro Koana, Pascal Kunz, and Rolf Niedermeier. Parameterized algorithms for colored clustering. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(4):4400–4408, Jun. 2023.

[35] Nicolas Klodt, Lars Seifert, Arthur Zahn, Katrin Casel, Davis Issac, and Tobias Friedrich. A color-blind 3-approximation for chromatic correlation clustering and improved heuristics. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, KDD '21, pages 882–891, New York, NY, USA, 2021. Association for Computing Machinery.

[36] Daniel Levi and David A Askay. *Group dynamics for teams*. SAGE publications, 2025.

[37] Pan Li and Olgica Milenkovic. Inhomogeneous hypergraph clustering with applications. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural*

Information Processing Systems 30 (NeurIPS 2017), pages 2308–2318. Curran Associates, Inc., 2017.

[38] Pan Li and Olgica Milenkovic. Submodular hypergraphs: p-laplacians, cheeger inequalities and spectral clustering. In *Proceedings of the 35th International Conference on Machine Learning (ICML 2018)*, pages 3014–3023. PMLR, 2018.

[39] Nate Veldt. Optimal lp rounding and linear-time approximation algorithms for clustering edge-colored hypergraphs, 2023.

[40] Qing Xiu, Kai Han, Jing Tang, Shuang Cui, and He Huang. Chromatic correlation clustering, revisited. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 26147–26159. Curran Associates, Inc., 2022.

[41] Justine Zhang, Cristian Danescu-Niculescu-Mizil, Christina Sauper, and Sean J. Taylor. Characterizing online public discussions through patterns of participant interactions. *Proc. ACM Hum.-Comput. Interact.*, 2(CSCW), November 2018.

[42] Dengyong Zhou, Jiayuan Huang, and Bernhard Schölkopf. Learning with hypergraphs: clustering, classification, and embedding. In *Proceedings of the 20th International Conference on Neural Information Processing Systems*, NIPS'06, pages 1601–1608, Cambridge, MA, USA, 2006. MIT Press.