The Black-and-White Coloring Problem on Tree-Structured Hypergraphs

Shira Zucker[1]

[1]Computer Science Department, Sapir Academic College, Israel

Abstract

The Black-and-White Coloring (BWC) problem seeks to find a partial vertex coloring of a graph (or hypergraph) that maximizes the number of white vertices for each number of black vertices, such that no black and white vertices are adjacent. This problem is NP-complete in general. In this paper, we present a polynomial-time algorithm to solve the BWC problem on a family of sparse hypergraphs whose hyperedge intersection graph forms a tree.

Keywords

combinatorial optimization, anticoloring, BWC, hypergraph, line graph

1. Introduction

The *Black-and-White Coloring (BWC) problem* on a graph is defined as follows. Given a finite connected undirected graph G and a positive integer b, find a partial coloring of G with exactly b black vertices and with w white vertices, where w is as large as possible and with the restriction that the black and white vertices are not adjacent. The hypergraph version of BWC is defined analogously: Given a hypergraph H and a positive integer b, find a partial coloring of H with exactly b black vertices and with w white vertices, where w is as large as possible and with the restriction that no hyperedge in E contains black and white vertices. The remaining vertices are left uncolored. Note that once the set of black vertices is fixed, the white vertices are determined, and the resulting coloring is deemed optimal. In this context, a solution is fully characterized by its set of black vertices.

The BWC problem has several practical applications. An example arises in chemical storage, where storage locations are represented by vertices and each hyperedge corresponds to a group of locations that must store mutually compatible chemicals. In this context, colors represent chemical types: A black vertex might indicate a location storing a reactive chemical, while a white vertex indicates a location with a nonreactive (safe) chemical. The BWC constraint ensures that no hyperedge (i.e., group of storage locations) contains incompatible chemicals, thereby enabling safe and efficient use of storage space. Another example arises in social network analysis, where individuals must be assigned to groups in a way that avoids conflicts of interest. For example, one group may represent members who support a particular initiative, while another group represents those who oppose it. The BWC model ensures that no individual in favor (black) is placed in direct interaction with an opponent (white), while neutral or undecided individuals can remain uncolored. This abstraction allows analysts to identify stable partitions of the network that minimize tension and highlight potential zones of agreement or conflict. For additional applications, see [1].

The problem was originated by Berge, who raised the following instance [6]. Given positive integers n and $b \leq n^2$, place b black and w white queens on an $n \times n$ chessboard, so that no black queen and white queen attack each other and with w as large as possible. Hansen *et al.* [10] formalized the general BWC problem and proved its NP-completeness while providing an $O(n^3)$ algorithm for trees. Berend and Zucker later improved this runtime to $O(n^2 \lg^3 n)$ for trees [2]. Related anticoloring problems based on rook placements [14] and king placements [1] have also been studied.

ICTCS 2025: Italian Conference on Theoretical Computer Science, September 10–12, 2025, Pescara, Italy
shiraz@sapir.ac.il (S. Zucker)

The Black-and-White Coloring problem can be naturally extended to settings involving more than two colors. This leads to the notion of an *anticoloring*, which is a partial vertex coloring of a graph with two or more colors such that no edge connects vertices of different colors. Formally, in the *general anticoloring problem*, we are given an undirected graph G and integers $b_1, \ldots, b_k$, and the task is to decide whether there exists a coloring in which exactly b_j vertices are assigned color j (for each $j = 1, \ldots, k$), while all other vertices may remain uncolored. Such a coloring is referred to as a $(b_1, \ldots, b_k)$-anticoloring. It was observed by [14] that this formulation can be conveniently expressed as an integer linear program. Heuristic methods, including tabu search [4] and a greedy probabilistic approach [5], have been proposed to solve the problem. In [16], the BWC problem was solved for chordal graphs. In addition, a two-approximation algorithm was presented in [15] to minimize erroneous edges in the full-coloring version (in which every vertex is colored black or white, while minimizing the number of edges connecting differently colored vertices).

A problem closely related to BWC is the *separation problem*. Here, one is given an n-vertex graph G and a constant $\alpha < 1$, and the goal is to partition the vertices of G into three sets A, B, C with the following properties:

(i) no edge has one endpoint in A and the other in B,
(ii) both A and B contain at most αn vertices, for $\alpha < 1$, and
(iii) the set C is relatively small.

The set C functions as a *separator* of G, since deleting it divides the graph into two subgraphs of bounded size. Using the vertices of A as black, those of B as white, and leaving C uncolored, one obtains a BWC of the graph (not necessarily an optimal one).

Typically, the separation problem is studied within specific graph classes $\mathcal{S}$ that are closed under taking subgraphs. An $f(n)$*-separator theorem* for $\mathcal{S}$ (see, e.g., [11]) asserts that every graph $G \in \mathcal{S}$ admits such a partition with $|C| \leq f(n)$.

In this paper, we extend the study of the BWC problem to hypergraphs. A hypergraph $H = (V, E)$ generalizes a graph by allowing each hyperedge $e \in E$ to be a subset of V. In the hypergraph version of the BWC problem, the restriction is that no hyperedge may contain both black and white vertices simultaneously. Our focus on sparse hypergraphs, where the weighted line graph forms a tree, allows us to leverage the tree structure for an efficient dynamic programming solution. Compared to previous work on trees, which achieved the complexities of time $O(n^3)$ [10] and $O(n^2 \lg^3 n)$ [2], our algorithm runs in $O(n^3)$, remarkably efficient given the increased complexity of hypergraphs compared to graphs. Note that such hypergraphs possess a small separator, similarly to trees and chordal graphs, facilitating efficient decomposition and dynamic programming. Our algorithm computes all pairs (b, w) representing optimal BWC's for a given hypergraph, providing a comprehensive solution for the optimization variant of the problem.

The remainder of the paper is organized as follows. In Section 2 we present some basic notions. Section 3 gives the main results. Section 4 details the proofs and describes the algorithm. Finally, Section 5 describes future research directions.

2. Basic Notions

We begin by recalling the notion of a (b, w)-coloring for the BWC problem. A (b, w)*-coloring* of a graph (or hypergraph) is a feasible partial coloring with exactly b black and w white vertices, where the constraints of the BWC problem are respected. A pair (b, w) is said to be *non-dominated* if there is no other feasible BWC solution that simultaneously uses at least b black vertices and at least w white vertices, while having a strictly larger total number of colored vertices. Any (b, w)-coloring corresponding to a non-dominated pair is termed an *optimal BWC*.

Our algorithm computes the set of all non-dominated pairs simultaneously. We will show how to use it to find the required BWC. The main result is summarized in Theorem 3.1.

Hypergraphs and the Weighted Line Tree. A hypergraph $H = (V, E)$ is defined as usual, with each hyperedge $e \in E$ being a subset of V.

We construct the weighted line graph $L(H) = (V_T, E_T)$ as follows:

1. Each vertex $h \in V_T$ corresponds to a hyperedge $e_h \in E$ and is assigned the weight $\nu(h) = |e_h|$.
2. Two vertices h and h' in $L(H)$ are adjacent (that is, $(h, h') \in E_T$) if and only if $e_h \cap e_{h'} \neq \emptyset$. The edge (h, h') is assigned the weight $\mu(h, h') = |e_h \cap e_{h'}|$.

Under the restriction that the hyperedge intersection pattern forms a tree, $L(H)$ is a tree and we refer to it as *weighted line tree*. Our approach first solves a modified full-coloring problem on this tree (in which every vertex is colored either black or white,) and then derives a feasible BWC for H. As usual, denote $|V| = n, |E| = m$. Note that if a vertex $v \in V$ is in three hyperedges e_1, e_2, e_3, then $e_1 \cap e_2 \neq \emptyset, e_2 \cap e_3 \neq \emptyset, e_1 \cap e_3 \neq \emptyset$, which forms a triangle in $L(H)$. Since $L(H)$ is a tree, this is impossible in our case, and therefore each vertex of H is contained in at most two hyperedges and therefore $m \leq 2n$.

Weighted Functions. For clarity, the weight functions of the vertices and edges of T are defined by:

$$\begin{aligned} &\nu(h) = |e_h|, && h \in V_T,\ e_h \in E, \\ &\mu(h, h') = |e_h \cap e_{h'}|, && (h, h') \in E_T,\ e_h, e_{h'} \in E \end{aligned} \tag{1}$$

Coloring Notions. In addition to a BWC of H, we introduce a *full-coloring* of the weighted line tree T. A full-coloring of T is a complete assignment of black or white to every vertex (without the constraint that adjacent vertices be of different colors). Given a full-coloring of T, a BWC of H is obtained by coloring each vertex $v \in V$ with the color assigned to each hyperedge containing v if all such hyperedges are uniformly colored; otherwise, v remains uncolored.

For each vertex h in the weighted line tree T, we store a field h.cList, which is a dictionary with two keys: *black* and *white*. The value h.cList[black] is the list B_h, containing non-dominated pairs (b, w) for the subtree rooted at h with h colored black, and h.cList[white] is the list W_h, containing non-dominated pairs (b, w) for the subtree rooted at h with h colored white. The content of this field for the root of T gives the desired non-dominated pairs for the hypergraph.

Figure 1 illustrates an example where a BWC with 6 black and 9 white vertices of a hypergraph is derived from a corresponding full-coloring of its weighted line tree. The numbers written inside each vertex h_i of the weighted line tree demonstrate the value $\nu(h_i) = |e_i|$. The weights of each edge (h_i, h_j) demonstrate the value $\mu(h_i, h_j) = |e_i \cap e_j|$.

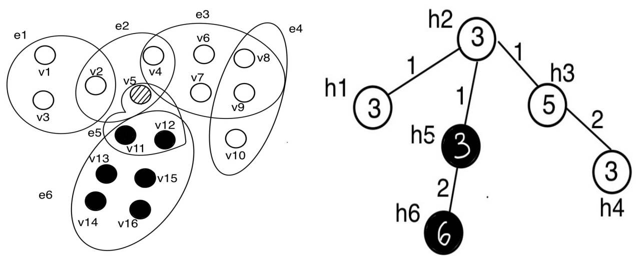

Figure 1: *left:* a BWC of a hypergraph, *right:* a corresponding full-coloring of its weighted line tree

3. Main Results

Theorem 3.1. *Algorithm 1 computes the complete list of non-dominated pairs for any hypergraph $H = (V, E)$ whose hyperedge intersection pattern forms a tree structure. The algorithm runs in $O(|V|^3)$ time.*

```
fullColorLineTree($H, T$)
Input: A hypergraph $H$ and its weighted line tree $T$ (with weight functions $\nu$ and $\mu$ defined in (1))
Output: The list optPairs of non-dominated pairs $(b, w)$ for $H$

$R \leftarrow$ root($T$)
for each leaf $h$ of $T$    // Initialization
    $h$.cList[black] $\leftarrow (\nu(h), 0)$
    $h$.cList[white] $\leftarrow (0, \nu(h))$
$R$.cList $\leftarrow$ solveLineTree($H, R$)    // Compute lists for internal nodes
optPairs $\leftarrow R$.cList[black] $\cup$ $R$.cList[white]
contract(optPairs)    // Remove dominated pairs
return optPairs
```

Algorithm 1: Compute all non-dominated pairs for a weighted line tree

Once the list of all non-dominated pairs is obtained, the maximal number of white vertices corresponding to any prescribed number b of black vertices can be determined by scanning for the pair (b', w) with minimal $b' \geq b$. We later discuss how to recover the actual BWC from the computed information.

Assume that T is arbitrarily rooted. For each vertex h of T, the two lists B_h and W_h are computed recursively. Algorithm 1 initializes these lists for the leaves and then invokes the recursive procedure in Algorithm 2 to compute the lists for the internal vertices. Two auxiliary routines, `extension` (Algorithm 3) and `merge` (Algorithm 4), are used to extend the lists when adding a new parent and to merge lists from different subtrees, respectively. Eventually, the algorithm uses the procedure `contract`, which gets a list and deletes all dominated pairs (as well as repeated occurrences of pairs). This procedure uses the bucket-sort algorithm (cf. [7]).

4. Proofs and Detailed Algorithm Description

In this section, we describe the algorithm in detail and prove its correctness. Throughout, $H = (V, E)$ denotes a hypergraph and $T = L(H) = (V_T, E_T)$ its weighted line tree. There is a natural correspondence between subtrees of T and subhypergraphs of H; that is, for a subtree T' with vertices corresponding to the hyperedges $e_1, \ldots, e_t$, the associated subhypergraph is induced by $e_1 \cup \cdots \cup e_t$.

4.1. From Full-Coloring to BWC

Although a full-coloring does not necessarily respect any BWC constraints of H, it provides the basis from which a feasible BWC is derived. For each vertex $v \in V$, if all hyperedges containing v receive the same color, then v is colored accordingly; otherwise, v is left uncolored. This procedure guarantees that no hyperedge contains both black and white vertices.

Let b and w denote the total number of black and white vertices in the resulting BWC. A full-coloring that produces the pair (b, w) will be called a (b, w)-full-coloring of T.

We will show later how to find a (b, w)-full-coloring of T from the output of Algorithm 1.

4.2. Recursive Computation on the Weighted Line Tree

Assume that T is arbitrarily rooted. For each vertex h of T, we maintain two lists, B_h and W_h, as explained before. These lists are stored in the field h.`cList`.

Algorithm 1 initializes the process and, after executing Algorithm 2, compiles the final list of non-dominated pairs. Algorithm 2 recursively computes this list. For each internal vertex R with children $r_1, r_2, \ldots, r_d$, it first obtains the lists for each child (using a post-order traversal), then calls the `extension` procedure (Algorithm 3) to update the lists by adding the parent R as the new root of the corresponding subtree, and finally merges these updated lists using the `merge` procedure (Algorithm 4).

```
solveLineTree($H$, $R$)
Input: A hypergraph $H$ and a root $R$ of its weighted line tree
Output: $R$.cList

if $R$ is a leaf
    return $R$.cList
$r_1, r_2, \ldots, r_d \leftarrow$ all children of $R$
for $i \leftarrow 1$ to $d$
    $r_i$.cList $\leftarrow$ solveLineTree($H, r_i$)
    list$_i$ $\leftarrow$ extension($H, r_i$.cList[black], ($r_i$.cList[white], $R$)

// Extend $T_{h_i}$ by adding $R$ as a parent
for $i \leftarrow 2$ to $d$
    list$_1$ $\leftarrow$ merge($H$, list$_1$[black], list$_i$[black], list$_1$[white], list$_i$[white])
$R$.cList $\leftarrow$ list$_1$
return $R$.cList
```

Algorithm 2: Generate cList for the subtree rooted at R

```
extension($H, B_r, W_r, R$)
Input: A hypergraph $H$; the lists $B_r, W_r$ for $r$ = root of a subtree; and the new parent $R$
Output: The updated lists $B_R, W_R$ for the subtree with root $R$, whose only child is r

$B_R, W_R \leftarrow$ empty list
for each $(b, w) \in B_r$
    append$(B_R, (b + \nu(R) - \mu(R, r),\ w))$
    append$(W_R, (b - \mu(R, r), w + \nu(R) - \mu(R, r)))$
for each $(b, w) \in W_r$
    append$(W_R, (b,\ w + \nu(R) - \mu(R, r)))$
    append$(B_R, (b + \nu(R) - \mu(R, r),\ w - \mu(R, r)))$
contract$(B_R)$    // Remove dominated pairs
contract$(W_R)$
return $B_R, W_R$
```

Algorithm 3: Extend a subtree by adding a new root

The extension procedure (Algorithm 3) takes as input the lists for the root of a subtree and 'lifts' the solution by introducing a new root R (which becomes the parent). The merge procedure (Algorithm 4) then combines the lists of two subtrees with the same new root.

The Contract function (Algorithm 5) removes dominated pairs to maintain only non-dominated pairs: This procedure uses the bucket-sort algorithm (cf. [7]).

Example 4.1. *Tables 1–6 show the performance of the algorithm on the weighted line tree of Figure 1. Table 1 gives the initial lists for the leaves h_1, h_4 and h_6. For example, h_1 corresponds to a hyperedge of size 3, so $B_{h_1} = (3, 0)$ (all vertices black) and $W_{h_1} = (0, 3)$ (all vertices white). Table 2 gives the resulting lists after performing Algorithm 3 on h_6.cList and h_4.cList to obtain h_5.cList and h_3.cList, respectively. For h_5, which has child h_6, the pair $(6, 0)$ in B_{h_6} is extended to $(7, 0)$ by adding $\nu(h_5) - \mu(h_5, h_6) = 3 - 2 = 1$ black vertex, and $(0, 6)$ in h_6 becomes $(0, 7)$. The pair $(0, 6)$ in W_{h_6} is extended to $(1, 4)$ in by adding $\nu(h_5) - \mu(h_5, h_6) = 1$ black vertex and subtracting $\mu(h_5, h_6 = 2)$ white vertices, since the vertices in $h_5 \cap h_6$ should be left uncolored. Tables 3,4 and 5 calculate h_2.cList step by step. Table 3 presents the temporary lists list$_1$, list$_2$, and list$_3$ for h_2, calculated by applying Algorithm 3 to the children h_1, h_5, and h_3, respectively. For example, list$_1$ corresponds to h_1 extended with h_2 as the new root, resulting in $(5, 0)$*

```
merge($H, B_{R^1}, B_{R^2}, W_{R^1}, W_{R^2}$)
**Input:** $B_{R^i}, W_{R^i}$: lists for the roots $R^i$, $i = 1, 2$, of the subtrees ($R^1$ and $R^2$ are copies of root R)
**Output:** $B_R, W_R$: the merged lists for the unified root $R$

$B_R, W_R \leftarrow$ empty list
**for each** $(b_1, w_1) \in B_{R^1}$
    **for each** $(b_2, w_2) \in B_{R^2}$
        append$(B_R, (b_1 + b_2 - \nu(R),\ w_1 + w_2))$
contract$(B_R)$    //delete dominated pairs
**for each** $(b_1, w_1) \in W_{R^1}$
    **for each** $(b_2, w_2) \in W_{R^2}$
        append$(W_R, (b_1 + b_2,\ w_1 + w_2 - \nu(R)))$
contract$(W_R)$    //delete dominated pairs
**return** $B_R, W_R$
```

Algorithm 4: Merge two subtrees with a common root

```
contract($L$)
**Input:** List $L$ of $(b, w)$ pairs
**Output:** List $L'$ containing only non-dominated pairs from $L$

**if** $|L| \leq 1$
    **return** $L$
Sort $L$ by $(b, w)$ lexicographically //sort by black count, then by white count
$L' \leftarrow \emptyset$
maxWhite $\leftarrow -1$
**for** $(b, w) \in L$ in sorted order //process in order to increase black count
    **if** $w >$ maxWhite
        $L' \leftarrow L' \cup \{(b, w)\}$
        maxWhite $\leftarrow w$
**return** $L'$
```

Algorithm 5: Contract dominated pairs from a list

Table 1
First step: Initializing

B_{h_1}	W_{h_1}	B_{h_6}	W_{h_6}	B_{h_4}	W_{h_4}
(3,0)	(0,3)	(6,0)	(0,6)	(3,0)	(0,3)

in the black list by adding $\nu(h_2) - \mu(h_2, h_1) = 3 - 1 = 2$ black vertices to $(3, 0)$, and $(2, 2)$ when h_1 is white, reflecting the intersection $h_2 \cap h_1$. Note that for clarity, we added a row named 'colored' to Table 3, which marks the vertices that are colored after performing Algorithm 3 (in contrast to the uncolored ones). These data can later be used to find the BWC itself. Table 4 shows the result of applying Algorithm 4 to combine list$_1$ and list$_2$ for h_2 before removing the dominated pairs, producing pairs like $(11, 0)$ by merging $(5, 0) \in$ list$_1$[black] and $(9, 0) \in$ list$_2$[black], subtracting $\nu(h_2) = 3$ to obtain $5 + 9 - 3 = 11$ black vertices. Table 5 then completes the computation of h_2.cList by merging the result from Table 4 with list$_3$, yielding the final non-dominated pairs such as $(10, 5)$, after eliminating dominated pairs like $(10, 4)$ that are superseded by pairs with more colored vertices.

Table 2
Second step: Computing h_5.cList and h_3.cList

B_{h_5}	W_{h_5}	B_{h_3}	W_{h_3}
(7,0),(1,4)	(0,7),(4,1)	(6,0),(3,1)	(0,6),(1,3)

Table 3
Third step: Temporary lists for h_2

list$_1$	black	(5,0)	(2,2)
	white	(0,5)	(2,2)
	colored	h_2	$h_2 - h_1$
list$_2$	black	(9,0),(3,4)	(2,6),(6,0)
	white	(0,9),(4,3)	(6,2),(0,6)
	colored	h_2	$h_2 - h_5$
list$_3$	black	(8,0),(3,3)	(2,5),(5,0)
	white	(0,8),(3,3)	(5,2),(0,5)
	colored	h_2	$h_2 - h_3$

Table 4
Computing h_2.cList by performing Algorithm 4 on the two first temporary lists

Lists before deleting dominated pairs	
B_{h_2}	(11,0),(5,4),(8,2),(2,6),(4,6),(8,0),(1,8)
W_{h_2}	(0,11),(4,5),(6,4),(0,8),(2,8),(6,2),(8,1)

Table 5
Computing h_2.cList by performing Algorithm 4 on the two last temporary lists

Lists of h_2 after deleting dominated pairs	
B_{h_2}	(16,0),(13,2),(11,3),(10,5),(9,6),(7,7),(6,8),(4,9)
W_{h_2}	(0,16),(2,13),(3,11),(5,10),(6,9),(7,7),(8,6),(9,4)

4.3. Correctness Proofs

We now briefly outline the proofs that establish the correctness of the algorithm.

Lemma 4.1. *The lists produced by Algorithm 3 on* r.`cList` *contain all non-dominated pairs for the subtree* T_r *extended by the new root* R.

Proof: Consider a subtree rooted at r with lists B_r and W_r, which, by induction, contain all non-dominated pairs for T_r. When extending to a new root R, we consider two cases for R's color.

Case 1: R is colored black.

- If r is also colored black, for each pair $(b, w) \in B_r$, the extended pair becomes $(b + \nu(R) - \mu(R, r), w)$. This is because the vertices in $e_R \setminus e_r$ are newly colored black, adding $\nu(R) - \mu(R, r)$ black vertices, while the vertices in $e_R \cap e_r$ remain black.
- If r is colored white, for each pair $(b, w) \in W_r$, the extended pair becomes $(b+\nu(R)-\mu(R, r), w-\mu(R, r))$. Here, vertices in $e_R \setminus e_r$ are colored black, but vertices in $e_R \cap e_r$ must be uncolored since r is white and R is black, so we subtract $\mu(R, r)$ from the white count.

Case 2: R is colored white.

- If r is also colored white, for each pair $(b, w) \in W_r$, the extended pair becomes $(b, w + \nu(R) - \mu(R, r))$, as vertices in $e_R \setminus e_r$ are newly colored white.
- If r is colored black, for each pair $(b, w) \in B_r$, the extended pair becomes $(b - \mu(R, r), w + \nu(R) - \mu(R, r))$. Vertices in $e_R \cap e_r$ must be uncolored due to conflicting colors (black in r, white in R), so we subtract $\mu(R, r)$ from the black count and add $\nu(R) - \mu(R, r)$ white vertices.

These updates ensure that all possible colorings of the extended subtree are considered, and the `contract` procedure removes any dominated pairs, leaving only the non-dominated ones.

Lemma 4.2. *If the lists for two subtrees rooted at R^1 and R^2 contain all non-dominated pairs for their respective subtrees, then Algorithm 4 produces the correct lists for their merge at R.*

Proof: Since R^1 and R^2 are copies of R, they correspond to the same hyperedge e_R. Therefore, the algorithm merges lists of the same color:

Case 1: Merging black lists B_{R^1} and B_{R^2}. For pairs $(b_1, w_1) \in B_{R^1}$ and $(b_2, w_2) \in B_{R^2}$, the merged pair is $(b_1 + b_2 - \nu(R), w_1 + w_2)$. The number $\nu(R)$ that is subtracted from the counting of black vertices represents the vertices belonging to the hyperedge R. Obviously, some of these vertices should be colored black in the BWC, but some of them should be left uncolored, as they belong to $e_R \cap h$, for some hyperedge h that is colored white in the full-coloring of $L(H)$. For each vertex in e_R that should be colored black in the corresponding BWC of the subhypergraphs associated with both trees rooted at R^1 and R^2, we need to subtract it to prevent double counting. We also need to subtract all the vertices in e_R that should be black only in the tree rooted at R^1 and not in the tree rooted at R^2 (or vice versa), as they should be left uncolored in the merged subtree. Note that since each $v \in V$ belongs to at most two hyperedges, the case that v is supposed to be left uncolored in both subtrees rooted at R^1 and R^2 is not possible. Together we subtract $\nu(R)$ vertices.

Case 2: Merging white lists W_{R^1} and W_{R^2}. Similarly, for pairs $(b_1, w_1) \in W_{R^1}$ and $(b_2, w_2) \in W_{R^2}$, the merged pair is $(b_1 + b_2, w_1 + w_2 - \nu(R))$, adjusting for overlapping white vertices in e_R and for vertices which are colored white in only one of the subtrees and uncolored in the second.

The algorithm does not merge across different colors (e.g., black and white lists), ensuring consistency in the color assignment to R. The `contract` procedure removes any dominated pairs from the merged lists.

Lemma 4.3. *For each vertex h of T, the calculated h.*`cList` *contains all non-dominated pairs corresponding to the subhypergraph induced by the subtree T_h.*

Proof: The proof is by induction on the height of the subtree T_h. For a leaf h, the initialization yields the correct lists. Assuming the correctness for all subtrees of height less than η, by Lemmas 4.1 and 4.2, Algorithm 3 and Algorithm 4 guarantee that the lists for a subtree of height η are correct.

4.4. Recovering the BWC Coloring

In order to find a (b, w)-full-coloring of T for a pair (b, w), we need to save some extra data during the running of Algorithms 3 and 4. For each pair computed by Algorithm 3, performed on the subtree rooted at h, we will record that h is colored black (white, respectively) if this pair was computed in B_h (W_h, respectively). For each pair computed using Algorithm 4, performed on the subtree rooted at h, we will record a link to the two pairs from which it was computed.

Algorithm 6 recovers the full-coloring itself, using backtracking and traversing the weighted line tree T top-down, starting from the root. Each vertex h will be colored with the color (black or white) chosen for it when computing the pair (b, w) in h.cList by Algorithm 3.

The second part of Algorithm 6 finds the BWC for H. For each $v \in V$, it checks the colors of all hyperedges containing v: if all are black, it colors v black; if all are white, the algorithm colors it white; otherwise, it leaves v uncolored.

4.5. Runtime Analysis

The construction of the weighted line tree takes $O(m^2 \cdot k)$, where $m = |E|$ is the number of hyperedges and k is the maximum size of a hyperedge [12]. Since $m \leq 2n$ and $k \leq n$, this construction takes $O(n^3)$.

The initialization of the leaves in Algorithm 1 takes $O(m) = O(n)$.

```
recoverFullColoring($H, T, (b, w)$)
Input: A hypergraph $H$, its weighted line tree $T$, and a target pair $(b, w)$
Output: A full BWC coloring of $H$

function backtrack($h$, $(b, w)$)
  if $(b, w) \in B_h$
    color[$h$] ← black
  else if $(b, w) \in W_h$
    color[$h$] ← white
  else
    $(b_1, w_1), (b_2, w_2)$ ← source pairs from fusion used to get $(b, w)$
    for each child $c$ of $h$
      backtrack($c$, $(b_i, w_i)$)    // use the matching pair for each child
end function

$R$ ← root($T$)
backtrack($R$, $(b, w)$)    // start from root and recover hyperedge colors in $T$
for each vertex $v \in V(H)$   // find the BWC for $H$
  if all hyperedges containing $v$ are black
    color[$v$] ← black
  else if all hyperedges containing $v$ are white
    color[$v$] ← white
  else color[$v$] ← uncolored
return color
```

Algorithm 6: Recover a full-coloring from stored dynamic programming tables

Algorithm 2 calls Algorithm 3 and Algorithm 4 for each of the m vertices of T, which is $O(n)$ since $m \leq 2n$. In algorithm 3, the input lists B_R, W_R each have a maximum size $O(n)$, since $b, w \leq n$, and the dominated pairs are removed. Each iteration processes a pair in $O(1)$ time. The procedure `contract` removes dominated pairs using a bucket sort in $O(N + n)$ time, where N is the input list size; here, $N = O(n)$, so this is $O(n)$, and the total per call is $O(n)$.

Algorithm 4 iterates over pairs in B_{R^1}, B_{R^2} (or W_{R^1}, W_{R^2}), which is $O(n^2)$, since each list has size $O(n)$. The `contract` procedure here takes $O(N + n)$ time with $N = O(n^2)$, so $O(n^2)$, and the total per call is $O(n^2)$.

Algorithm 2 makes $O(n)$ calls to Algorithm 3, totaling $O(n^2)$, and $O(n)$ calls to Algorithm 4 across all merges (since T has $m - 1 = O(n)$ edges), totaling $O(n^3)$. Combining construction ($O(n^3)$) and initialization ($O(n)$), the overall runtime is $O(n^3)$.

Note that this runtime is for finding the list of all non-dominated pairs.

In order to find a full-coloring (and after that the required BWC), we need to save data during computations as explained in Section 4.4. Algorithm 6, which performs that, works on each vertex of T in a constant time. Recall that each vertex of the hypergraph is contained in at most two hyperedges, therefore, the second part of the algorithm is also linear. We find that the runtime of the algorithm is still $O(n^3)$.

5. Future Work

Our algorithm can be applied in scenarios like network partitioning, where hyperedges represent groups of nodes that must be uniformly colored (e.g., assigning servers to compatible tasks). Future work could implement and test the algorithm on real-world datasets, such as chemical storage hypergraphs from compatibility databases or social network group structures, to evaluate its practical performance and scalability.

Future research may extend our results to more general hypergraph classes such as Berge-acyclic hypergraphs and other forms of hypertrees. Note that the weighted line graph of a Berge-acyclic hypergraph is not necessarily a tree, so different techniques will be required. In addition, further work may focus on improving the runtime of the algorithm (e.g., similarly to what was suggested in [2]) and exploring practical heuristics for larger instances.

Declaration on Generative AI

During the preparation of this work, the author(s) used ChatGPT, Gemini, and overleaf in order to grammar and spelling check, paraphrase and reword. After using these tools, the author(s) reviewed and edited the content as needed and assume(s) full responsibility for the content of the publication.

References

[1] D. Berend, E. Korach, and S. Zucker, Anticoloring of a family of grid graphs, *Discrete Optimization*, 5(3):647–662, 2008.

[2] D. Berend and S. Zucker, The Black-and-White coloring problem on trees, *Journal of Graph Algorithms and Applications*, 13(2):133–152, 2009.

[3] D. Berend, E. Korach, and S. Zucker, A Reduction of the Anticoloring Problem to Connected Graphs, *Electronic Notes in Discrete Mathematics*, 28:445–451, 2006.

[4] D. Berend, E. Korach and S. Zucker, Tabu Search for the BWC Problem, Journal of Global Optimization: 54/4:649–667, DOI: 10.1007/s10898-011-9783-1, 2012.

[5] D. Berend and S. Mamana, A Greedy Probabilistic Heuristic for Graph Black-and-White Anticoloring, *Journal of Graph Algorithms and Applications*, 18(1):1–14, 2024.

[6] C. Berge, *Hypergraphs: Combinatorics of Finite Sets*, North-Holland, 1989.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press and McGraw-Hill, 1990.

[8] P. Damaschke, On an Ordering Problem in Weighted Hypergraphs, *Combinatorial Algorithms: 32nd International Workshop*, IWOCA 2021, Proceedings Pages 252 - 264.

[9] J. Erickson, Lower bounds for linear satisfiability problems, *Chicago Journal of Theoretical Computer Science*, 1999(8).

[10] P. Hansen, A. Hertz, and N. Quinodoz, Splitting trees, *Discrete Mathematics*, 165(6):403–419, 1997.

[11] R. J. Lipton and R. E. Tarjan, A Separator Theorem for Planar Graphs, *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[12] X. T. Liu, et al, Parallel Algorithms and Heuristics for Efficient Computation of High-Order Line Graphs of Hypergraphs, *arXiv:2010.11448*, 2020.

[13] R. E. Tarjan, Efficiency of a Good But Not Linear Set Union Algorithm, *Journal of the ACM*, 22:215–225, 1975.

[14] O. Yahalom, Anticoloring Problems on Graphs, M.Sc. Thesis, Ben-Gurion University, 2001.

[15] S. Zucker, An Approximation Algorithm for the BWC Problem, in *Proceedings of the 17th International Conference on Mathematical Methods in Science and Engineering (CMMSE'17)*, pp. 2063–2066, 2017.

[16] S. Zucker, The Black-and-White Coloring Problem on Chordal Graphs, *Journal of Graph Algorithms and Applications*, 16(2):261–281, 2012.