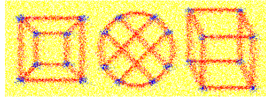

Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 16, no. 2, pp. 261–281 (2012)

The Black-and-White Coloring Problem on Chordal Graphs

Shira Zucker

Department of Software Systems, Sapir Academic College, Shaar Hanegev, Israel

Abstract

Given a graph G and positive integers b and w, the black-and-white coloring problem asks about the existence of a partial vertex-coloring of G, with b vertices colored black and w white, such that there is no edge between a black and a white vertex. This problem is known to be NP-complete in general. We provide here a polynomial time algorithm for chordal graphs.

Submitted: September 2009	Reviewed: January 2011	Revised: May 2011	Reviewed: August 2011
Revised: September 2011	Accepted: February 2012	Final: March 2012	Published: March 2012
Article type: Regular paper		Communicated by: M. Furer	

E-mail address: zuckers@cs.bgu.ac.il (Shira Zucker)

1 Introduction

The *Black-and-White Coloring (BWC) problem* is defined as follows. Given an undirected graph G and positive integers b, w, determine whether there exists a partial coloring of G such that b vertices are colored black and w vertices white (with all other vertices left uncolored), such that no black vertex and white vertex are adjacent. Such a partial coloring, if exists, is a *Black-and-White Coloring (BWC)* of the graph.

One application of the BWC problem comes from the chemical industry. A set of b samples of a product B and w samples of a product W has to be stored in $n \geq x + y$ different available places. For security reasons, due to the chemical nature of the samples and the configuration of the storing places, there are certain pairs of places that cannot contain two different types of products. The question is whether it is possible to store all samples by respecting these restrictions. By constructing a graph G that has a vertex for each storing place and an edge between each two places that are not allowed to contain two different types of products, this problem reduces to the BWC problem.

Another application is solved explicitly in [2]: Items of two data types, D_1 and D_2, are stored in a 2-dimensional table. A person would like to retrieve data of type D_1 or of type D_2, but not of both. When retrieving data, we would like to allow a one-unit error in each of the table's indexes. In case of an error, we do not want a person trying to retrieve an element of type D_1 to extract an element of type D_2. An additional goal is to populate the elements of type D_1 in a way that will maximize the number of places left in the table for the elements of type D_2.

We sometimes refer to the optimization version of this problem, in which we are given a graph G and a positive integer b, and have to color b of the vertices black, so that there will remain as many vertices as possible which are non-adjacent to any of the b vertices. These latter vertices are to be colored white, and the resulting coloring is *optimal*. Clearly, when referring to a BWC, it suffices to refer to its black vertices only.

For example, in Figure 1, an optimal BWC with $B = 3$ and $W = 4$ is depicted. Notice that by coloring black three out of the four vertices on the right of the figure, one would obtain a BWC with $B = 3$ and $W = 3$, which is not optimal.

The problem was originated by Berge, who raised the following instance [15].

Problem 1.1 *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.*

The BWC problem has been introduced in general, and proved to be NP-complete, by Hansen *et al.* [15]. In the same paper, an $O(n^3)$ algorithm for trees was given. In [3], an $O(n^2 \lg^3 n)$ algorithm for trees was given. Kobler *et al.* [16] gave a polynomial algorithm for partial k-trees with a fixed k.

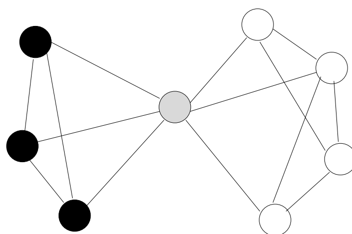

Figure 1: An optimal BWC with $B = 3$ and $W = 4$

Yahalom [20] investigated an analogous problem to that suggested by Berge, using rooks instead of queens, and gave a sub-linear algorithm to this problem. For special cases, in which the ratio between the sides of the board is an integer or close to an integer, she derived an explicit formula for the optimal solution. In [2], we investigated an analogous problem, using kings instead of queens, and provided explicit optimal solutions for the toroidal and the non-toroidal versions.

In [5] we examined several heuristic algorithms, in particular tabu search, for solving the problem in general.

The BWC problem admits a generalization for any number of colors. An *anticoloring* of a graph is a partial vertex coloring with two or more colors, in which no two adjacent vertices have distinct colors. In the general *anticoloring problem*, we are given an undirected graph G and positive integers $b_1, \ldots, b_k$, and have to determine whether there exists an anticoloring of G such that b_j vertices are colored in color j, $j = 1, \ldots, k$. We call such an anticoloring a $(b_1, \ldots, b_k)$-anticoloring. Yahalom [20] noticed that it is easy to rewrite the anticoloring problem as an integer linear programming problem.

The BWC problem is closely related to the *separation problem.* In the latter, we are given an n-vertex graph G and a constant $\alpha < 1$, and have to partition the vertices of G into three sets A, B, C such that (i) no edge joins a vertex in A with a vertex in B, (ii) A and B contain at most αn vertices each, and (iii) C is "small". The set C is a *separator* of G, i.e., its removal splits the graph into two parts, each with at most some fixed fraction of the vertices. This fraction is defined by α. Thus, coloring the vertices of A black and those of B white, and leaving those of C uncolored, we obtain a BWC of G.

The separation problem usually deals with a class $\mathcal{S}$ of graphs, closed under the subgraph relation. An $f(n)$-*separator theorem* (cf. [17]) for $\mathcal{S}$ is a theorem which ensures that every graph G in $\mathcal{S}$ may be split as above with $|C| \leq f(n)$.

A graph is *chordal* if every cycle of length at least 4 has a chord, i.e., an edge connecting two nonconsecutive vertices on the cycle. Clearly, any induced subgraph of a chordal graph is chordal as well.

It was shown in [14] that any chordal graph which has no $(k + 2)$-clique can be separated with $f(n) = k$ and $\alpha = 1/2$ in time $O(m\alpha(m, n))$, where m is the number of edges in the graph and $\alpha(m, n)$ is a very slowly growing function described in [19]. Thus, by [4], we can easily obtain an algorithm

which finds, for such a graph with n vertices, a BWC with b black vertices and $w \geq n - b - O(\min\{k\log n, b\})$ white vertices, in the same runtime. However, this algorithm does not necessarily find an optimal BWC.

In this paper we give an algorithm which solves the BWC problem for chordal graphs in time $O(\chi n^3)$, where χ is the chromatic number of the graph. We will show that, if n is large, then our algorithm works in $O(\chi n^2)$ time for almost all chordal graphs. Note that the family of chordal graphs is much larger than that of trees, and contains roughly $2^{n^2/4}$ graphs on n vertices [1].

Sections 2 and 3 present the main results. In Section 4 we survey briefly some results concerning chordal graphs, which are used in the sequel. The proofs and the algorithms are detailed in Sections 5, 6 and 7. Section 8 explains how a BWC for given b and w can actually be found. In Section 9 we extend our algorithm to solve the anticoloring problem with many colors.

The author is grateful to D. Berend for helpful comments and suggestions.

2 Main Results

A (b, w)-*coloring* of G is a BWC of G with b black and w white vertices. The pair (b, w) is *non-dominated* if there exists no BWC with $b' \geq b$ black and $w' \geq w$ white vertices, and $b' + w' > b + w$; otherwise, it is *dominated*. A (b, w)-coloring for a non-dominated pair (b, w) is an *optimal BWC*. Our algorithm solves simultaneously the BWC problem for all pairs (b, w).

Theorem 1 *Algorithm 1 finds the list consisting of all non-dominated pairs of a chordal graph G with n vertices in time $O(n^4)$.*

3 Improvement of the Algorithm

The following theorem improves the runtime of Algorithm 1 for chordal graphs whose chromatic number is $o(n)$.

Theorem 2 *Let χ be the size of the maximum clique of the graph. Algorithm 1 can be improved to run in time $O(\chi n^3)$.*

Chordal graphs are known to be perfect. Therefore, the size of the maximum clique of a chordal graph is equal to its chromatic number.

Remark 3 ***(a)*** *As we shall see in the proof of Theorem 2, if the number of internal vertices of the clique tree (see Definition 2 below) of G is k, then we obtain a runtime of $O(\chi n^2 k \lg \frac{n}{k})$ in the theorem. For $k = o(n)$, this yields an improvement.*

(b) *From the proof of Theorem 2 we also get that, if the number of maximal cliques of size $\Omega(n)$ is $O(1)$, then Theorem 2 holds, where χ is now the maximal size of the cliques of size $o(n)$.*

Chordal graphs may well contain cliques of size $\Omega(n)$. For such graphs, Algorithm 1 runs in $O(n^4)$ time. How does Algorithm 1 perform on a typical chordal graph? Unfortunately, by [1], the largest clique in most chordal graphs is of size about $n/2$. Fortunately, though, most chordal graphs (i.e., a proportion of at least $1-(\frac{1}{2}\sqrt{3}+\epsilon)^n$ for sufficiently large n) are split [1], in which case we attain an improved result.

A *split graph* is a graph whose vertices can be partitioned into two sets, one of which induces a clique and the other an independent set. Obviously, every split graph is chordal.

Theorem 4 *Algorithm 1 may be modified to run in time $O(\chi n^2)$ for split graphs.*

Once the list of all non-dominated pairs has been found, it is straightforward to find the maximal number of white vertices in a BWC of G for any prescribed number b of black vertices. In fact, search the list for the element (b', w) with minimal b' such that $b' \geq b$. The w in that pair is the required number.

Throughout the rest of this paper, we deal only with connected chordal graphs. It follows from [4] that, after solving the problem for such graphs, the solution for all chordal graphs is easy to obtain.

4 Chordal Graphs – Preliminaries

Let G be a chordal graph and $a, b \in V(G)$. A set $S \subset V(G)$ is a *minimal ab-separator* if the graph induced by $V(G) - S$ contains no path between a and b, and no proper subset of S is such. S is a *minimal vertex separator* if it is a minimal ab-separator for some vertices a, b.

Theorem 5 *[9] A graph G is chordal if and only if every minimal vertex separator of G is a clique.*

An example for the next two definitions can be found in Figure 2 below.

Definition 1 *[7] Let $G = (V, E)$ be a chordal graph. The* `weighted clique intersection graph` *(or simply* `clique graph`*) of G is denoted by $C(G) = (V_C, E_C, \mu)$, with $\mu : E_C \to \mathbf{N}$, and given by:*

1. *Its vertex set V_C is the set $\{K_1, K_2, \ldots, K_m\}$ of all maximal cliques in G.*
2. *$E_C = \{(K_i, K_j) : K_i, K_j \in V_C,\ K_i \cap\ K_j \neq \emptyset\}$.*
3. *$\mu(K_i, K_j) = |K_i \cap K_j|$, $(K_i, K_j) \in E_C$.*

Recall that, by [13], a chordal graph contains at most $|V|$ maximal cliques, and therefore $|V_C| \leq |V|$.

The following notion will play a critical role in the paper.

Definition 2 *Let $G = (V, E)$ be a chordal graph and $C(G)$ its clique graph. A* `clique tree` *in G is a maximum weighted spanning tree of $C(G)$.*

Blair *et al.* [7] present a linear time algorithm for finding a clique tree of a chordal graph.

The following result is due to Lundquist [18].

Theorem 6 *Let T be a clique tree of a chordal graph $G = (V, E)$. A set $S \subset V$ is a minimal vertex separator of G if and only if $S = K \cap K'$ for some $(K, K') \in E(T)$.*

For example, in Figure 2 we have that $v_7 = K_2 \cap K_3$ and $\{v_2, v_3\} = K_1 \cap K_2$ are two different minimal vertex separators of the given chordal graph.

For every clique tree T, consider the set consisting of all sets $K \cap K'$ with $(K, K') \in E(T)$. By Theorem 6, each element in this set is a minimal vertex separator of G.

The following theorem was proved in [6].

Theorem 7 *Given a clique tree T of a chordal graph, for any pair of distinct cliques $K, K' \in V(T)$, the set $K \cap K'$ is contained in every clique on the path connecting K and K' in T.*

For example, in Figure 2(b) we can see that $K_1 \cap K_4 = \{v_2\}$, and indeed, $v_2 \in K_2$, which is a clique on the path connecting K_1 and K_4.

5 Proof of Theorem 1

5.1 Sketch of the Algorithm

Throughout the next two sections G will denote a chordal graph and T a clique tree of G, constructed as in [7]. Note that there is a correspondence between subtrees of T and certain subgraphs of G, whereby for each subtree T' there exists a corresponding subgraph G', induced by $K_1 \cup \ldots \cup K_t$, where $V(T') = \{K_1, K_2, \ldots, K_t\}$.

The main steps of the algorithm, which will be detailed in Section 5.2, are as follows.

In general, the algorithm takes a chordal graph G, computes its clique tree T and finds in T the list of all non-dominated pairs (b, w) such that G admits a BWC with b black and w white vertices.

In order to find all the non-dominated pairs, we begin by defining a new notion for trees, called full-coloring. Each full-coloring of a clique tree T implies a corresponding BWC for the corresponding chordal graph G.

According to the algorithm, each vertex in the tree is attached with two lists, depending on its color, black or white. Each list of each vertex contains pairs (b, w) saying that the subtree rooted at that vertex has a (b, w)-full-coloring. The corresponding (original) graph has a BWC with b black and w white vertices. These lists are computed in a post-order form, from the leaves of the tree to its root, using two aid procedures: merge and extension.

Eventually, to actually find a BWC with given numbers b,w of black and white vertices respectively, find a pair (b', w') such that $b' \geq b$ and $w' \geq w$ and color the graph as explained in Section 8.

5.2 The Algorithm

Lemma 3 *Given an optimal BWC of G, the set U of uncolored vertices satisfies $U = \bigcup_{(K,K') \in E_0} (K \cap K')$ for some $E_0 \subseteq E(T)$.*

Proof: Obviously, U separates G into $m \geq 2$ connected components $G_1, \ldots, G_m$. For $1 \leq i < j \leq m$, let $U_{ij} \subseteq U$ be a minimal subset of U that separates the components G_i and G_j. Obviously, $U = \bigcup_{1 \leq i < j \leq m} U_{ij}$, and each U_{ij} is a minimal ab-separator of G for any $a \in V(G_i)$ and $b \in V(G_j)$. By Theorem 6, for each i and j, $U_{ij} = K_i \cap K_j$ for two cliques K_i and K_j such that $(K_i, K_j) \in E(T)$. □

It will be convenient to introduce another notion of coloring of (clique) trees. A *full-coloring* of T is a coloring of the vertices of T, such that each vertex is colored either black or white. We emphasize that there are no constraints on the coloring of adjacent vertices, so that a full-coloring is in general neither a coloring nor a BWC. Note that T is both vertex- and edge-weighted, where the weight functions $\nu : V(T) \to \mathbf{N}$ and $\mu : E(T) \to \mathbf{N}$ are given by

$$\begin{array}{ll} \nu(K) = |K|, & K \in V(T), \\ \mu(K_1, K_2) = |K_1 \cap K_2|, & (K_1, K_2) \in E(T). \end{array} \tag{1}$$

Given a full-coloring of T, we construct a BWC of G as follows. For each vertex $v \in V(G)$, consider all maximal cliques in G containing it. If all these cliques have the same color in T, color v in that same color; otherwise, leave v uncolored. Since, for any two adjacent vertices in G, there exists a maximal clique containing them both, the resulting coloring is indeed a BWC. Let b and w be the number of black and white vertices, respectively, in this BWC. The given full-coloring of T thus gives rise to a (b, w)-coloring of G, and will therefore be referred to as a (b, w)-full-coloring of T. Note that, if (b, w) happens to be a non-dominated pair for G, then every (b, w)-coloring of G can be obtained from some full-coloring of T. Thus, we refer to (b, w) also as a non-dominated pair for T. A (5,6)-coloring of a chordal graph, obtained from a (5,6)-full-coloring of a corresponding clique tree is presented in Figure 2.

Assume T is rooted (arbitrarily). To each vertex $K \in V(T)$ we attach two lists, B_K and W_K. The former list consists of all the non-dominated pairs for the subtree of T rooted at K, where K is constrained to be colored black. W_K is the analogous list for the case where K is constrained to be colored white. The variable K.dList consists of these two lists.

To simplify the algorithm, we split it into several algorithms, called by each other.

Algorithm 1, which finds all the non-dominated pairs, initializes the lists B_K and W_K for all leaves of T, and then invokes Algorithm 2, which in turn finds these lists for the internal vertices of T. In particular, by unifying the two lists attached to the root of T, we obtain the required output, as stated in Theorem 1. Eventually, the algorithm uses the procedure `contract`, which gets a list and deletes from it all dominated pairs (as well as repeated occurrences of pairs). This procedure uses the bucket-sort algorithm (cf. [8]).

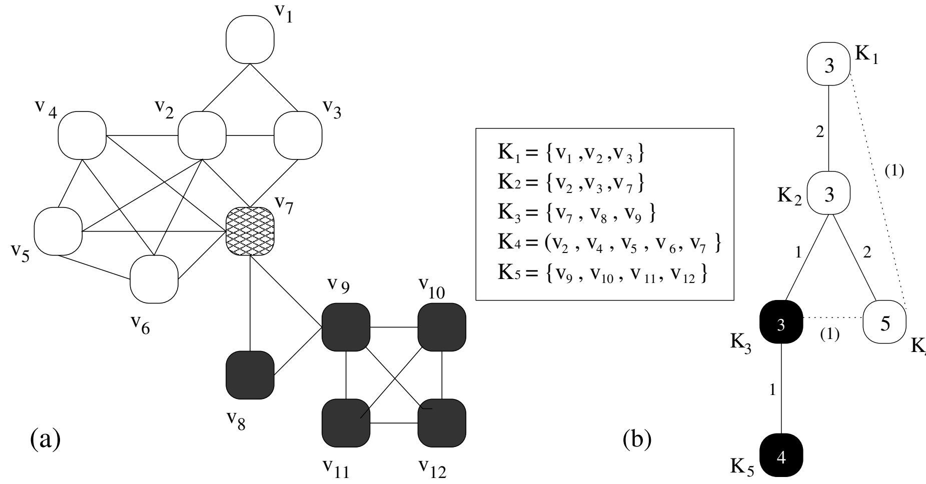

Figure 2: (a) a BWC of a chordal graph, (b) a corresponding full-coloring of its clique tree (where the dotted edges belong only to the clique graph).

fullColorTree(G, T)
Input: A chordal graph G and a clique tree T thereof (ν and μ are as in (1))
Output: The list optPairs, providing the non-dominated pairs (b, w) for T

$R \leftarrow$ root(T)
for each leaf K of T //initialization
  K.dList[`black`] $\leftarrow (\nu(K), 0)$
  K.dList[`white`] $\leftarrow (0, \nu(K))$
R.dList $\leftarrow$ solveCliqueTree(G, R) // find the lists for the internal vertices
optPairs $\leftarrow R$.dList[`black`] $\cup R$.dList[`white`]
contract(optPairs) // delete dominated pairs
return optPairs

Algorithm 1: Find all non-dominated pairs for a clique tree.

Algorithm 2 below is based on the algorithm that solves the BWC problem on trees [15]. It traverses the tree in post-order and computes the lists for each vertex. Let d be the degree of the root R. At the beginning the algorithm invokes Algorithm 3 on the i-th child K_i of R, $1 \leq i \leq d$, (with added edge between R and K_i) in order to find the list for the subtree induced by $\{R\} \cup V(T_{K_i})$. This list is saved as list$_i$. Namely, list$_i$[`black`] (respectively, list$_i$[`white`]) is the current

list obtained for the case where the root is colored black (respectively, white). After finding the lists corresponding to all the children of R, the algorithm merges all these lists step by step. At each step i, the algorithm performs Algorithm 4 on list_1 and list_i. The merged list is saved as list_1. Eventually, list_1 is the required list for T_R.

Algorithm 2 invokes Algorithms 3 and 4, which perform the computations. Algorithm 4 is performed on pairs of dLists, until it merges them all to a single one.

```
solveCliqueTree(G, R)
Input: A chordal graph G and a root R of its clique tree
Output: R.dList

if R is a leaf
  return R.dList
K_1, K_2, ..., K_d ← all children of R
for i ← 1 to d
  K_i.dList ← solveCliqueTree(G, K_i)
  list_i ← extension(G, K_i.dList) // list for the subtree T_{K_i}, adding R
                                   // as root
for i ← 2 to d  // find the lists for the tree rooted at R
  list_1 ← merge(G,list_1[black],list_i[black],list_1[white],list_i[white])
R.dList ← list_1
return R.dList
```

Algorithm 2: Generate dList for the root of a clique tree.

Algorithm 3 is given root(T).dList for some subtree T and finds root(T').dList, where T' is composed of T and a new root, R', whose only child is root(T). R' is in fact the father of R in T.

The computation of the algorithm is very simple. Let R =root(T) and R' =father(R). For each $(b, w) \in B_R$ it records the pair $(b+\nu(R')-\mu(R', R), w)$ in $B_{R'}$, and $(b - \mu(R', R), w + \nu(R') - \mu(R', R))$ in $W_{R'}$. Note that in the first case, where R and R' are both black, all the vertices of $R' \subseteq G$ are colored black. In the second case, where R is black and R' is white, all the vertices of $R' \cap R \subseteq G$ must be left uncolored. Similarly, for each $(b, w) \in W_R$ the algorithm records the pair $(b, w+\nu(R')-\mu(R', R))$ in $W_{R'}$ and $(b+\nu(R')-\mu(R', R), w-\mu(R', R))$ in $B_{R'}$. Eventually, it uses the procedure `contract` to delete dominated pairs.

Note that after performing Algorithm 3, Algorithm 2 calls Algorithm 4 to merge two dLists. The performance of Algorithm 4 requires specific input (for example, the exact sets of colored vertices), which is computed in Algorithm 3. More specifically, in addition to computing the required lists, Algorithm 3 records, for each pair (b, w) in the two lists of a vertex $K \in V(T)$, all

the vertices $v \in K$ that are colored in the corresponding (b, w)-coloring. These data are saved by Algorithm 3 in (b, w).colored. The procedure `append` adds a given pair at the end of a given list. The last pair in a list l is l.tail.

```
extension(G, B_R, W_R, R')
Input: A chordal graph G, the lists B_R, W_R, where R = root(T), and the
       vertex R' = father(R)
Output: B_R', W_R' – the lists for R' = root(T'), where T' is composed
        of T and a new root R', whose only child is R

B_R', W_R' ← empty list
for each (b, w) ∈ B_R
   append(B_R', (b + ν(R') − μ(R', R), w))
   B_R'.tail.colored ← R'    // all v ∈ R' ⊆ G are colored
   append(W_R', (b − μ(R', R), w + ν(R') − μ(R', R)))
   W_R'.tail.colored ← R' − R    // all v ∈ R' ∩ R must be left uncolored
for each (b, w) ∈ W_R
   append(W_R', (b, w + ν(R') − μ(R', R)))
   W_R'.tail.colored ← R'    // all v ∈ R' are colored
   append(B_R', (b + ν(R') − μ(R', R), w − μ(R', R)))
   B_R'.tail.colored ← R' − R    // all v ∈ R' ∩ R must be left uncolored
contract(B_R') // delete dominated pairs
contract(W_R')
return B_R', W_R'
```

Algorithm 3: Add a new root to a given subtree.

Algorithm 4 is given the lists for two subtrees having a common root, and finds the lists for the root of the subtree obtained by merging these two subtrees. For each $(b_1, w_1) \in B_{R^1}$ (respectively, W_{R^1}) and each $(b_2, w_2) \in B_{R^2}$ (respectively, W_{R^2}), the algorithm computes $(b_1 + b_2 - \texttt{size}, w_1 + w_2)$ (respectively, $(b_1 + b_2, w_1 + w_2 - \text{size})$), where the variable `size` is the number of colored vertices in the unified root, and appends it to the list B_R (respectively, W_R). Note that the colored vertices of $R \subseteq G$ are exactly the intersection of the colored vertices of $R^1 \subseteq G$ and $R^2 \subseteq G$. Similarly to Algorithm 3, we record these data. After finding the required lists, all dominated pairs are deleted, using the procedure `contract`.

Example 8 *In Tables 1–6 below the performance of the algorithm on the clique tree of Figure 2 is shown. In Table 1 we give the lists for the leaves of T. Table 2 gives the resulting lists after performing Algorithm 3 on K_5.dList to obtain K_3.dList. In Table 3 we see the temporary lists list$_1$ and list$_2$, which are the input for Algorithm 4. In Tables 4 and 5 we give the results of the*

```
merge(G, B_{R^1}, B_{R^2}, W_{R^1}, W_{R^2})
Input: B_{R^i}, W_{R^i} – the lists for the roots R^i, i = 1, 2, of the two subtrees
Output: B_R, W_R – The lists for the unified root R

B_R, W_R ← empty list
for each (b_1, w_1) ∈ B_{R^1}
  for each (b_2, w_2) ∈ B_{R^2}
    size ← |(b_1, w_1).colored ∪ (b_2, w_2).colored| // the number of colored
                                                    //vertices in the unified root
    append(B_R, (b_1 + b_2 − size, w_1 + w_2))
    B_R.tail.colored ← (b_1, w_1).colored∩(b_2, w_2).colored
contract(B_R) // delete dominated pairs
for each (b_1, w_1) ∈ W_{R^1}
  for each (b_2, w_2) ∈ W_{R^2}
    size ← |(b_1, w_1).colored ∪ (b_2, w_2).colored|
    append(W_R, (b_1 + b_2, w_1 + w_2 − size))
    W_R.tail.colored ← (b_1, w_1).colored∩(b_2, w_2).colored
contract(W_R) // delete dominated pairs
return B_R, W_R
```

Algorithm 4: Merge two subtrees with a common root.

performance of Algorithm 4 before and after the deletion of dominated pairs. In Table 6 we give the final list, obtained after performing Algorithm 3 on K_2.dList, which was obtained in the preceding two tables.

5.3 Conclusion of the Proof of Theorem 1

Let T_v be the subtree of T rooted at v, and $G(T_v)$ be the subgraph of G corresponding to T_v.

The following lemmas prove the correctness of the algorithm. Suppose first that R_1 and R_2 are two children of R. Let T_R^i be the subtrees of T defined by

$$T_R^i = V((T_{R_i} \cup \{R\}), E(T_{R_i}) \cup \{(R, R_i)\}), \quad i = 1, 2,$$

both rooted at R.

Lemma 4 *The lists obtained by performing Algorithm 3 on R_i.dLists, $i = 1, 2$, consist of all the non-dominated pairs for $G(T_R^i)$.*

Proof: Assume, say, that R_i is colored black. For each non-dominated pair (b, w) for $G(T_{R_i})$, by coloring R black, we add exactly $|R| - |R \cap R_i| = \nu(R) - \mu(R, R_i)$ black vertices and do not change the number of white vertices. Note that, if

B_{K_4}	W_{K_4}	B_{K_5}	W_{K_5}
(5,0)	(0,5)	(4,0)	(0,4)

Table 1. First step: Initializing.

B_{K_3}	W_{K_3}
(6,0),(2,3)	(0,6),(3,2)

Table 2. Second step: Computing K_3.dList.

	`black`	(8,0),(4,3)	(2,5),(5,1)
list_1	`white`	(0,8),(3,4)	(5,2),(1,5)
	colored	K_2	$K_2 - K_3$
	`black`	(6,0)	(1,3)
list_2	`white`	(0,6)	(3,1)
	colored	K_2	$K_2 - K_4$

Table 3. Third step: Temporary lists for K_2.

Lists before deleting dominated pairs		
B_{K_2}	(11,0),(7,3),(5,5),(8,1),(6,3),(2,6)	(1,8),(4,4)
W_{K_2}	(0,11),(3,7),(5,5),(1,8),(3,6),(6,2)	(8,1),(4,4)
`size`	3	2

Table 4. Computing K_2.dList by performing Algorithm 4 on the temp lists.

B_{K_2}	(11,0),(7,3),(5,5),(8,1),(2,6),(1,8)
W_{K_2}	(0,11),(3,7),(5,5),(1,8),(6,2),(8,1)

Table 5. Perform contract(K_2.dList) to delete dominated pairs.

B_{K_1}	(12,0),(8,3),(6,5),(9,1),(3,6)
W_{K_1}	(0,12),(3,8),(5,6),(1,9),(6,3)

Table 6. Last step: Computing K_1.dList by performing Alg. 3 on K_2.dList.

there exists a vertex K such that $(R, K) \in E_C$ and $(R_1, K) \in E(T)$, then, according to Theorem 7, we get $R \cap K \subseteq R_1 \cap K$, and therefore the color of K does not influence this computation. By coloring R white, we add exactly $|R| - |R \cap R_i| = \nu(R) - \mu(R, R_i)$ white vertices and subtract $|R \cap R_i| = \mu(R, R_i)$ black vertices (which should be left uncolored). Again, by Theorem 7, if there exists a vertex K as above, then $R \cap K \subseteq R \cap R_1$ and we may ignore the color of K. □

Lemma 5 *If we have the lists for each subtree T_R^i, $i = 1, 2$, then by performing Algorithm 4 on these lists, we get the required lists for $G(T_R)$.*

Proof: We split the proof into two cases.

Case 1: R_1 and R_2 are colored in the same color.

The black (and white) vertices in $G(T_R)$ are the same as in $G(T_R^i)$, $i = 1, 2$, except for the vertices which were turned to uncolored in Algorithm 3. Thus, if the number of black (respectively, white) vertices in $G(T_{R_i})$ is b_i (respectively, w_i), $i = 1, 2$, then the total number of black (respectively, white) vertices is $b_1 + b_2 - \texttt{size}$ (respectively, $w_1 + w_2 - \texttt{size}$), where `size` is equal to $|(b_1, w_1).\text{colored} \cup (b_2, w_2).\text{colored}|$.

Case 2: R_1 and R_2 are colored in different colors.

Besides the considerations described in Case 1, we need to check that, if $R_1 \cap R_2$ is non-empty, then it is left uncolored. Assume, say, that R_1 is colored black and R_2 white. Since T is a tree, $(R_1, R_2) \notin E(T)$. Consider the path $R_1 R R_2$ in T. Assume, say, that R is colored black. Then, by the algorithm, $R \cap R_2$ is left uncolored. By Theorem 7 we have $R_1 \cap R_2 \subseteq R \cap R_2$, which concludes the proof. □

Lemma 6 *For each vertex K of T, K.dLists consists of all the non-dominated pairs for $G(T_K)$.*

Proof: We prove this lemma by induction on the height of T_K. If the height is zero, then T_K consists of K only, and the algorithm gives K.dList[`black`]$=(\nu(K), 0)$ and K.dList[`white`]$=(0, \nu(K))$, which are the required lists for $G(T_K)$.

Assume the lemma is correct for all subtrees with height up to $h-1$. Let T_K be a subtree of height h, and let $K_1, K_2, \ldots, K_d$ be the children of K. Obviously, the heights of $T_{K_1}, T_{K_2}, \ldots, T_{K_d}$ are at most $h - 1$, and the lemma is correct for them. By Lemma 4, for each T_K^i we have the required lists. Applying the merge procedure over and over, on two children at a time, by Lemma 5, we get the required lists for T_K. □

5.4 Runtime of the Algorithm

The construction of a clique graph takes linear time (cf. [7]). Finding a clique tree is done by Prim's algorithm for finding a minimal spanning tree of a graph in $O(|E_C| \lg |V_C|)$ time, where $G_C = (V_C, E_C)$ is the clique graph. The procedure `contract` has a linear runtime. Therefore, the runtime of Algorithm 1 is identical to the runtime of Algorithm 2 with the addition of $O(|E_C| \lg |V_C|)$ time.

Algorithm 2 calls Algorithm 3 exactly once for each vertex, and Algorithm 4 exactly d times for each vertex with d children in the clique tree. Thus, it calls both algorithms $O(n)$ times.

For each pair in K.dList (out of at most $2n$ pairs), for some vertex K, Algorithm 3 performs computations in $O(1)$ time and then records all the vertices of some clique in $O(n)$ time. Therefore the total runtime of Algorithm 3 is $O(n^2)$.

Each of the double loops in Algorithm 4 is performed over $O(n^2)$ indexes. Each computation of `size` in the loops takes $O(n)$ time. Therefore, the total runtime of Algorithm 4 is $O(n^3)$.

Thus, the total running time of Algorithm 1 is $O(n^4)$. $\square$

6 Proof of Theorem 2

In this section we prove Theorem 2. The bottleneck of our algorithm is Algorithm 4. In this section we reduce the runtime of the latter to $O(\chi n^2)$. (Recall that χ is the chromatic number of the graph.) Thus, by choosing a suitable order for the performances of Algorithm 4 on each vertex, the total runtime of Algorithm 1 decreases to $O(\chi n^2 k \lg(n/k))$, where k is the number of internal vertices in T.

We begin with

Lemma 7 *Given a clique $K \in V(T)$ and the list of non-dominated pairs obtained by Algorithm 1 for K, consider for each pair (b, w) the corresponding (b, w)-coloring of the subgraph G' corresponding to T_K. The number of uncolored vertices $v \in K$ in this coloring is at most $\sum_{K_1 \text{ child of } K} |K \cap K_1|$.*

Proof: Let $v \in K$ be an uncolored vertex for a specific non-dominated pair. Obviously, $(v, u_1) \in E$ and $(v, u_2) \in E$, where u_1 is colored black and u_2 is colored white, and $u_i \in K_i, i = 1, 2$, for some children K_1, K_2 of K. It suffices to prove that $v \in K \cap K_1$ or $v \in K \cap K_2$. Assume this is not the case. Then, there exists $K' \in V(T)$ such that $v, u_1 \in K'$. Thus, $v \in K \cap K'$ and $u_1 \in K_1 \cap K'$. Since $K \cap K' \neq \emptyset$, we get that $(K, K') \in E_C$. Therefore, T contains one of the paths $p_1 = KK_1K'$ and $p_2 = K_1KK'$. If T contains p_1, then according to Theorem 7, since $v \in K \cap K'$, we get that $v \in K \cap K_1$, which ends the proof. If T contains p_2, then according to Theorem 7, since $u_1 \in K_1 \cap K'$, we get that $u_1 \in K \cap K'$, and therefore $u_1 \in K$.

Equivalently, we get that $u_2 \in K$. Now, u_1 is black and u_2 is white, and it is impossible for them both to belong to the same clique K, in contradiction to our assumption. This proves the lemma. $\square$

Our purpose is to compute `size` in Algorithm 4 faster. Recall that, for each two pairs (b_1, w_1) and (b_2, w_2) in the lists of the roots R^1 and R^2 of the subtrees to be merged,

$$\texttt{size} = |(b_1, w_1).\text{colored} \cup (b_2, w_2).\text{colored}|\,.$$

Note that, denoting by R the root of the unified tree, we have

$$\texttt{size} = \nu(R) - |(b_1, w_1).\text{uncolored} \cap (b_2, w_2).\text{uncolored}|\,.$$

Therefore, instead of recording for each pair (b_i, w_i) the colored vertices, we rather record the uncolored vertices. These vertices are maintained in a sorted list, sorted by the vertices' names.

Thus, in order to find the value of `size`, simply go over the two sorted lists of R^1 and R^2 in parallel, to find the number of vertices which are left uncolored in both lists. This is detailed in Algorithm 5. Note that Algorithm 5 comes instead of line 4 in Algorithm 4.

```
findSize((b1,w1).uncolored,(b2,w2).uncolored,ν(R))
Input: The lists of the uncolored vertices of the two pairs from
       Algorithm 4 and the weight of the unified root R
Output: The variable size required for Algorithm 4

size ← 0
h1 ← (b1,w1).uncolored.head
h2 ← (b2,w2).uncolored.head
while h1.next ≠NULL and h2.next ≠NULL
  if h1.name < h2.name
    h1 ← h1.next
  else if h2.name < h1.name
    h2 ← h2.next
  else if h1.name = h2.name
    size←size+1
    h1 ← h1.next
    h2 ← h2.next
size ← ν(R) − size
return size
```

Algorithm 5: Find the variable `size` quickly.

In order to record the list of uncolored vertices, both in Algorithm 3 and in Algorithm 4, we want to record (b_1, w_1).uncolored $\cup(b_2, w_2)$.uncolored in a sorted way. Similarly to Algorithm 5, we run over the two input lists in parallel, but this time we record each vertex which appears in at least one of these lists. Since the lists are already sorted, the new list is also sorted.

The runtime of this procedure is proportional to that of Algorithm 5, and therefore to the number of uncolored vertices in the two lists. By Lemma 7, the number of uncolored vertices in the list of R^i, $i = 1, 2$, is at most

$$\sum_{R' \text{ child of } R^i} \left| R^i \cap R' \right| .$$

Therefore, the runtime of Algorithm 4 is

$$O(n^2 \cdot (\sum_{R_c^1 \text{ child of } R^1} \left| R_c^1 \cap R^1 \right| + \sum_{R_c^2 \text{ child of } R^2} \left| R_c^2 \cap R^2 \right|)),$$

and since R^1 and R^2 are both instances of R, this is equal to

$$O(n^2 \cdot \sum_{R' \text{ child of } R^1 \cup R^2} |R' \cap R|).$$

Now, if d_i is the degree of a vertex K_i and $\chi_i = |K_i|$, then Algorithm 4 is performed $d_i - 1$ times on K_i. The total runtime of all these performances depends on the sequence of $d_i - 1$ times it is performed, and is thus equal to $O(n^2 \cdot \sum_{K \text{ child of } K_i} |K \cap K_i| \cdot X)$, where X is the number of times that extension(G, B_K, W_K, K_i) is inserted as an input to the algorithm. Such a sequence of merges can be represented by a binary tree M, with the children of K_i as leaves. Every internal node of M corresponds to one merge, and X is equal to the depth of K in M. Therefore, if we perform Algorithm 4 in the order specified by Algorithm 2, we get that $X = O(d_i)$, and the total runtime of Algorithm 4 on a vertex K_i is $O(n^2 \chi_i d_i)$. By performing Algorithm 4 each time on the i-th and the $(i+2^k)$-th children, $0 \leq k \leq \lceil \lg d \rceil, 1 \leq i \leq d - 2^{k+1} + 1$ and $i = c \cdot 2^{k+1}$ for some constant c, as if M is as balanced as possible, we get that $X = O(\lg d_i)$, and the total runtime on a vertex becomes $O(n^2 \chi_i \lg d_i)$.

Similarly, the total runtime of Algorithm 3 on K_i is $O(n \chi_i \lg d_i)$.

Thus, the total runtime of Algorithm 1 is

$$O(n^2 \sum_{i=1}^{n} \chi_i \lg d_i) = O(\chi n^2 \sum_{i=1}^{n} \lg d_i),$$

which is maximized when all d_i's are equal. Since $\sum_{i=1}^{n} \lg d_i \leq n - 1$, the total runtime is $O(\chi n^2 k \lg (n/k))$, where k is the number of internal vertices in T.

Thus, if T has $\Theta(n)$ internal vertices, the runtime is $O(\chi n^3)$, but if it has $o(n)$ internal vertices, we obtain a better result. This concludes the proof of Theorem 2.

Let us now explain the second part of Remark 3. If the number of maximal cliques K with $\nu(K) = \Omega(n)$ is some constant C, and the size of all other maximal cliques is bounded by $Y = o(n)$, then it is easy to show that the total runtime of the algorithm will be $C \cdot O(n^3) + O(n) \cdot O(Y n^2) = O(Y n^3)$.

7 Proof of Theorem 4

Let G be a split graph [1]. It is easy to show that there exists a clique tree T of G in which all the leaves are children of the root, and the root is the maximal clique of G (see Figure 3). In fact, such a clique tree can be found by taking the maximal clique of the graph as the root, and then taking all other maximal cliques (each of which consists of a single vertex from the independent set and all its neighbors) as its children. The values of ν and μ are as described in Section 5.

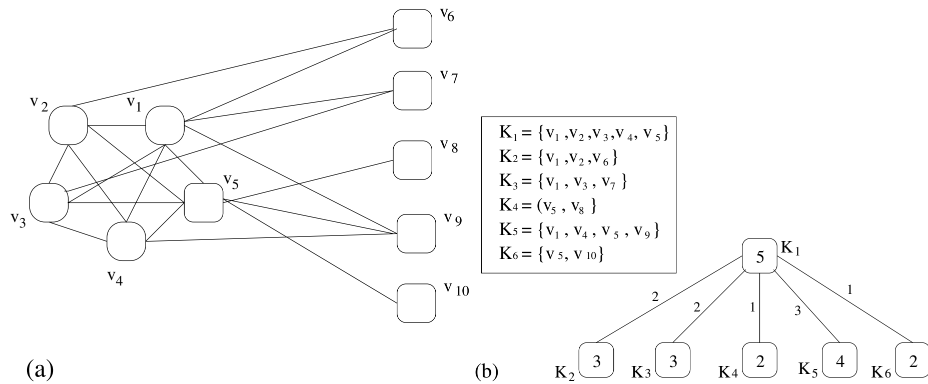

Figure 3: (a) a split graph, (b) a clique tree of the split graph.

Lemma 8 *If the clique tree T in Algorithm 1 is of height 1, then the algorithm runs in time $O(\chi n^2)$, where χ is the chromatic number of the chordal graph.*

Proof: For each leaf L, the algorithm initializes each of the lists B_L and W_L with a list of size 1. The computation of the required list for the root starts with performing Algorithm 3 on each of the leaves, which creates lists of size 2. Finally, Algorithm 4 gets each time two lists as input: one of size at most n and the other, of size exactly 2. Therefore, the runtime of the improved Algorithm 4 becomes $O(\chi n)$, and thus the runtime of Algorithm 1 is reduced to $O(\chi n^2)$. □

Theorem 4 follows straightforwardly from Lemma 8.

A linear-time algorithm for recognizing split graphs is presented in [11]. This algorithm also finds the maximum clique in the graph, which will immediately give us the desired clique tree.

8 Finding a BWC

Suppose we are required to actually find a BWC of a chordal graph G with given numbers b, w of black and white vertices, respectively. We may assume, without loss of generality, that (b, w) belongs to the list optPairs found by Algorithm 1.

In fact, find a pair (b', w') in optPairs such that $b' \geq b$, $w' \geq w$. (If no such pair exists, then there is no BWC as required.) After constructing a BWC with b' black and w' white vertices, we uncolor $b' - b$ black and $w' - w$ white vertices.

We first construct a full-coloring of the clique tree T from the input of Algorithm 1. To this end, we change Algorithm 4 to save pointers from each non-dominated pair it finds to the non-dominated pairs it was obtain from. The constructed full-coloring of T gives rise to a BWC of G, as described in Section 5.2.

9 Extension to Many Colors

In this section we discuss the anticoloring problem with k colors for a constant k.

The extension of the results of Section 2 to this general case is simple, and we shall explain it briefly.

Instead of non-dominated pairs, we have here non-dominated k-tuples. To each vertex we attach k lists, where the i-th list, $1 \leq i \leq k$, consists of all non-dominated k-tuples for the case it is colored i.

Algorithm 1 needs to be updated to initialize the leaves of the clique tree with the appropriate k-tuples. Its output is now the union of the k lists of the root. In Algorithm 2, the only change is the input of Algorithm 4, which should contain all k lists, instead of only two.

More modifications are required for Algorithms 3 and 4. First we denote by L_{i_R} the i-th list of a vertex R. Since the number of optimal k-tuples is at most n^{k-1}, this is the maximal possible length of each list L_{i_R}. For convenience, in the input to the append procedure, called by Algorithm 6, we write only the values for (b_i, b_j) (or (b_i) in case $i = j$), since the other values remain unchanged. This is not the case for the call to the same procedure in Algorithm 7.

The computation of the runtime of Algorithms 1 and 2 does not change. The runtime of Algorithm 6 is $O(k^2 n^k)$ and that of Algorithm 7 is $O(kn^{2k-1})$. Since k is a constant, the total runtime of Algorithm 1 is $O(n^{2k})$.

Note that since the number of k-tuples might be n^{k-1}, this is also the lower bound for the optimal runtime, and our algorithm cannot be improved drastically.

```
multiColorExtension(G, L_{1_R}, ..., L_{k_R}, R')
Input: A chordal graph G, the lists L_{i_R}, 1 ≤ i ≤ k where R = root(T),
       and the vertex R' = father(R)
Output: L_{1_R'}, ..., L_{k_R'} – the lists for R' = root(T'), where T' is
        composed of T and a new root R', whose only child is R

L_{1_R'}, ..., L_{k_R'} ← empty list
for i = 1 to k
  for each (b_1, ..., b_k) ∈ L_{i_R}
    for j = 1 to k
      if j == i
        append(L_{i_R'}, (b_i + ν(R') − μ(R', R)))
        L_{i_R'}.tail.colored ← R'
      else
        append(L_{j_R'}, (b_i − μ(R', R), b_j + ν(R') − μ(R', R)))
        L_{i_R'}.tail.colored ← R' − R
for i = 1 to k
  contract(L_{i_R'})
return L_{1_R'}, ..., L_{k_R'}
```

Algorithm 6: Add a new root to a given subtree.

```
multiColorMerge(G, L_{1_R^1}, ..., L_{k_R^1}, L_{1_R^2}, ..., L_{k_R^2})
Input: L_{i_R^1}, L_{i_R^2} – the lists for the roots R^j, j = 1, 2, of the two
       subtrees where 1 ≤ i ≤ k
Output: L_{1_R}, ..., L_{k_R} – The lists for the unified root R

L_{1_R}, ..., L_{k_R} ← empty list
for i = 1 to k
  for each (b^1_1, ..., b^1_k) ∈ L_{i_R^1}
    for each (b^2_1, ..., b^2_k) ∈ L_{i_R^2}
      size ← |(b^1_1, ..., b^1_k).colored ∪ (b^2_1, ..., b^2_k).colored|
      append(L_{i_R}, (b^1_1 + b^2_1, ..., b^1_i + b^2_i − size, ..., b^1_k + b^2_k))
  contract(L_{i_R})
return L_{1_R}, ..., L_{k_R}
```

Algorithm 7: Merge two subtrees with a common root.

References

[1] E. A. Bender, L. B. Richmond and N. C. Wormald, Almost all chordal graphs split, *J. Austral. Math. Soc.*, A/38:214–221, 1985.

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

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

[4] 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.

[5] D. Berend, E. Korach and S. Zucker, Heuristic algorithms for the BWC problem, 2007, submitted.

[6] P. A. Bernstein and N. Goodman, Power of natural semijoins, *SIAM J. Comput.* 10:751–771, 1981.

[7] J. R. S. Blair and B. W. Peyton, An introduction to chordal graphs and clique trees, *Graph Theory and Sparse Matrix Computations*, 56/1-30, Springer Verlag, 1993.

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

[9] G. A. Dirac, On rigid circuit graphs. *Abh. Math. Sem. Univ. Hamburg* 25:71-76, 1961.

[10] J. Erickson, Lower bounds for linear satisfiability problems, *Chicago J. Theoret. Comput. Sci.*, 1999(8).

[11] M. C. Golumbic, Algorithmic graph theory and perfect graphs, *Academic Press*, 1980.

[12] P. Galinier, M. Habib and C. Paul, Chordal graphs and their clique graphs, *Graph-Theoretic Concepts in Computer Science*, WG'95, volume 1017 of LNCS, pages 358–371, 1995

[13] F. Gavril, The intersection graphs of subtrees in trees are exactly the chordal graphs, *J. of Comb. Theory*, B/16:47-56, 1974.

[14] J. R. Gilbert, D. J. Rose and A. Edenbrandt, A separator theorem for chordal graphs, *SIAM J. Alg. Disc. Meth.* **5** 306–313, 1984.

[15] P. Hansen, A. Hertz and N. Quinodoz, Splitting trees, *Disc. Math.*, 165/6:403–419, 1997.

[16] D. Kobler, E. Korach and A. Hertz, On black-and-white colorings, anticolorings and extensions, preprint.

[17] R. J. Lipton and R. E. Tarjan, A separator theorem for planar graphs, *Appl. Math.* **36/2** 177–189, 1979.

[18] M. E. Lundquist, Zero patterns, chordal graphs and matrix completions, PhD thesis, Clemson University, 1990.

[19] R. E. Tarjan, Efficiency of a good but not linear set union algorithm, *Journal of the ACM*, 22:215–225, 1975.

[20] O. Yahalom, Anticoloring problems on graphs, M.Sc. Thesis, Ben-Gurion University, 2001.