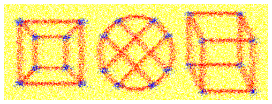

Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 13, no. 2, pp. 133–152 (2009)

The Black-and-White Coloring Problem on Trees

Daniel Berend[1] *Shira Zucker*[2]

[1]Departments of Mathematics and Computer Science,
Ben-Gurion University, Beer Sheva 84105, Israel
[2]Department of Computer Science,
Ben-Gurion University, Beer Sheva 84105, 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. We suggest an improved algorithm for solving this problem on trees.

Submitted:	Reviewed:	Revised:	Accepted:
June 2008	February 2009	March 2009	March 2009
	Final: April 2009	Published: June 2009	
Article type: Regular paper		Communicated by: H. Meijer	

E-mail addresses: berend@cs.bgu.ac.il (Daniel Berend) 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 vertex-coloring of G such that b vertices are colored black and w vertices in white (with all other vertices left uncolored), such that no black vertex and white vertex are adjacent.

One application of the BWC problem is to the problem of storing chemical products, where certain pairs of places cannot contain different products. (For other applications see, for example, [1].)

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 in 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*. Note that it may well be the case that, given an optimal BWC, we can increase the number of black vertices without decreasing the number of white vertices. Clearly, when referring to a BWC, it suffices to refer to its black vertices only.

The BWC problem has been introduced and proved to be NP-complete by Hansen *et al.* [5]. In the same paper, a polynomial algorithm for trees was given. A polynomial algorithm for partial k-trees with a fixed k was suggested by Kobler *et al.* [7]. Yahalom [11] gave a sub-linear algorithm for the graphs obtained by the moves of a rook on a chessboard. For an $m \times n$ board, this is in fact the Cartesian product (cf. [10]) $K_m \Box K_n$ of two complete graphs. In [1], we provided explicit optimal solutions for the graphs obtained by the moves of a king on a chessboard. Note that, for an $m \times n$ board, this is in fact the strong product (cf. [3]) $P_m \boxtimes P_n$ of two simple paths.

The algorithm for trees, suggested by Hansen *et al.*, has running time of $O(n^3)$. In this paper we introduce another algorithm, whose running time is $O(n^2 \lg^3 n)$. We also present an improvement to our algorithm, which works for almost all labeled trees in time $n^{1+o(1)}$. Note that we simultaneously find, for all $b \in [0, n]$, the corresponding optimal w.

In Section 2 we present formally the problem and state the main theorems. In Section 3 we find it convenient to pose a more general version of our problem. The rest of that section is devoted to the solution of the generalized problem. In Section 4 we present the algorithm which actually colors the given tree. Section 5 presents possible improvements of our algorithm.

The authors are grateful to A. Melkman for drawing their attention to [6].

2 Main Results

Let T be a tree with n vertices. The attributes of a BWC of T are given by a pair (b, w), in which b is the number of black vertices and w the number of white vertices. Thus, by having an array containing the optimal w for each value of b, we identify the attributes of all optimal BWCs. We refer to a BWC with b black

and w white vertices as a (b, w)-*coloring*.

Problem 2.1

Input: *A rooted n-vertex tree T.*

Output: *An array $maxW$ which, for each $0 \leq b \leq n$, gives the maximal w such that there exists a (b, w)-coloring of T.*

Theorem 1 *Problem 2.1 is solved by Algorithm 1 in time $O(n^2 \lg^3 n)$.*

Theorem 2 *Given a tree T and two integers b and w, Algorithm 4 finds a (b, w)-coloring for T in time $O(n^2 \lg^3 n)$.*

Our proof hinges on the following lemma.

Lemma 1 *For any optimal BWC of a tree T on n vertices, the number of uncolored vertices is at most $\lg n$.*

Going over the proof, one readily verifies that, if the upper bound $\lg n$ in the lemma was reduced, we could improve the runtime in Theorems 1 and 2. Specifically, if $\lg n$ could be reduced to some quantity $g(n) = o(\lg n)$, then the runtime could be reduced to $O(n^2 g^2(n) \lg n)$. However, after performing some experiments on complete binary trees, we believe that, in the worst case, the bound of $\lg n$ in Lemma 1 cannot be significantly reduced.

In some cases we can significantly improve Theorems 1 and 2. Given an n-vertex tree T, a number $b \in [1, n-1]$ is *economical* if an optimal BWC with b black vertices contains exactly one uncolored vertex.

Proposition 3 *For almost all labeled trees on n vertices:*

1. *The set of all economical b's can be found in time $n^{1+o(1)}$.*
2. *Given an economical b, an optimal BWC with b black vertices can be found in linear time (after the pre-processing of part 1).*

Recall that there exist n^{n-2} labeled trees on n vertices. The term 'almost all trees' means all trees but a fraction that tends to 0 as n grows.

By Proposition 3, given a tree T and an integer b, we may do the following: If the tree is such that all economical b's are easily found (which happens when the maximal degree of a vertex of T is relatively small; see the proof of Proposition 3), then we first check if the given b is economical; if so, we can find an optimal BWC in a much more efficient way than that of Theorem 2. If T is not such, or the given b is not economical, we revert to the algorithm of the theorem.

We performed several tests (see Section 5) in order to investigate the percentage of the number of economical b's. We found that, when increasing n, the percentage of economical b's decreases. However, this decrease is very slow; for example, for $n = 1000$ there are on the average about 469 economical b's. We also noticed that small and large b's have good chances of being economical, whereas b's around $\frac{n}{2}$ are mostly non-economical. Our experiments are discussed in detail in Section 5.

3 Proof of Theorem 1

3.1 A Recursive Approach and a more General Problem

We assume that the input tree T is rooted. For each vertex v, denote by T_v the subtree rooted at v.

To each vertex v of T we attach a $3 \times (|T_v| + 1)$-table, v.maxWhite, where entry (c, b) contains the maximal w for which there exists a (b, w)-coloring of T_v, with v colored in c. Here c ranges over the set {`black`, `white`, `uncolored`}. Thus v.maxWhite[`black`], for example, contains for each value b the maximal possible w, assuming that v is colored black.

Our algorithm starts by finding a vertex v, which separates the tree into several subtrees of relatively small sizes. One of these subtrees is the tree obtained from T by removing all vertices of T_v (if v is not the root of T). All other subtrees are rooted at the children of v. (In fact, it will be more convenient for us to adjoin v to each of the latter subtrees, so that they are all rooted at v.) The algorithm works recursively on each of these smaller subtrees, and finds the required table for each. The table of T_v is calculated iteratively by taking the union of larger and larger subtrees of T_v, two at a time.

The vertex v separating the tree may be any vertex which is not a leaf. To fix ideas, it will be more convenient (although not crucial) for us to select this vertex as one which separates the tree into as small as possible subtrees. The existence of such a vertex follows from the following well-known theorem [12].

Theorem 4 *Any n-vertex tree can be split in linear time into several connected components, each with at most $\frac{n}{2}$ vertices, by removing a single vertex.*

Remark 3.1 *By [2], Theorem 4 gives rise to a linear time approximation algorithm, which finds a BWC with $w \geq w_{\text{opt}} - \lg n$ white vertices, where w_{opt} is optimal corresponding to the given b.*

In the course of the algorithm, after the table attached to the subtree rooted at some vertex has been found, we do not need that subtree any more. Rather, the root of the subtree becomes a leaf, and all other vertices are disposed. However, this new leaf carries with it some non-trivial information, namely the table corresponding to the subtree rooted at it. Thus, at later stages we need to deal with instances of the problem in which leaves are equipped with attached data, which may be of size $O(n)$. Thus, we replace our original problem by the following, more general one. Denote by $L(T)$ the set of leaves of T.

Problem 3.1

Input: *A rooted n-vertex tree T, with a $3 \times (m_v + 1)$-table v.maxWhite attached to each leaf v (the m_v's being arbitrary positive integers). For $c \in$ {`black`, `white`, `uncolored`} and $0 \leq b \leq m_v$, the number v.maxWhite[c][b] is the maximal w such that, by coloring v in c, it is possible to be "awarded" with b black and w white vertices.*

Output: *A table root(T).maxWhite which, for each* $c \in \{$`black`, `uncolored`, `white`$\}$ *and* $0 \leq b \leq n + \sum_{v \in L(T)}(m_v - 1)$*, provides the maximal* w *such that there exists a* (b, w)*-coloring of* T *with root(*T*) colored* c*.*

In this problem, the table v.maxWhite may be intuitively thought of as providing the attributes of BWCs of some (perhaps unknown to us) subtree rooted at v with m_v vertices. (This subtree was eventually replaced by its root v in the course of the recursion.) Thus, for example, each entry (c, b) is at most $m_v - b$. Moreover (see Lemma 2 below), we assume that $b + v.\text{maxWhite}[c][b] \geq m_v - \lg m_v - 2$. Note that, for each vertex v, the entries $v.\text{maxWhite}[\texttt{black}][0]$ and $v.\text{maxWhite}[c][m_v]$ for $c \neq \texttt{black}$, are meaningless and should be ignored.

Theorem 5 *Problem 3.1 is solved by Algorithm 2 in time* $O(nm \lg^3 m)$*, where* $m = \max\{n, \sum_{v \in L(T)} m_v\}$.

In the course of the performance of the algorithm on the original problem, some subtrees are reduced to single vertices, namely their root becomes a leaf of the current tree. At the moment a subtree T_v is reduced, v is provided with a table of size $3 \times (m_v + 1)$, where $m_v = |T_v| + \sum_{y \in L(T_v)}(m_y - 1)$. This table is attached to v until some subtree containing v will also be reduced and its root will become a new leaf, at which point the table will be disposed. Notice that, for each vertex v, we have $m_v = 1 + \sum_{y \in Y} m_y$, where Y is the set of children of v. The runtime of Algorithm 2 depends, besides the number of vertices, on the size of the tables attached to the leaves. Therefore, it depends on two parameters, n and m.

In order to find the table root(T).maxWhite, which provides the attributes of all optimal BWCs of T, we invoke Algorithm 1. This algorithm transforms the given instance of the problem into an instance of Problem 3.1 and then invokes Algorithm 2 to actually find the required table.

3.2 The Algorithm

We begin with

Proof of Lemma 1: The proof is by induction on n. The case $n = 1$ is trivial. Assume the lemma is true for all trees with less than n vertices, and let T be a tree on n vertices. Let C be an optimal BWC of T with B black vertices. We will find a BWC with B black vertices in which there are at most $\lg n$ uncolored vertices. Therefore, the optimal coloring C also contains at most $\lg n$ uncolored vertices. By Theorem 4, there exists a separation vertex, splitting T into several connected components, each with at most $\frac{n}{2}$ vertices. Go over the components in some order, and color black all vertices of each component whose size is less than the number of vertices still to be colored black. Denote by b the number of vertices colored black after this stage. Obviously, each component which is not colored black contains more than $B - b$ vertices. Therefore, by the induction hypothesis, coloring $B - b$ vertices black in an optimal way in one of these components whose size is, say, x produces at most $\lg x \leq \lg \frac{n}{2}$ uncolored vertices. Thus, the number of uncolored vertices in T is at most $1 + \lg \frac{n}{2} = \lg n$. □

Lemma 2 *For any optimal BWC of a tree T and any vertex v, the number of uncolored vertices in the subtree T_v is at most* $\lg |T_v| + 2$.

Proof: Let C be an optimal BWC of T in which some subtree T_v contains more than $\lg |T_v| + 2$ uncolored vertices. Let b_v be the number of vertices in T_v which are colored black by C. By Lemma 1, there exists a coloring C_v of T_v with $b_v + 1$ black and at most $\lg |T_v|$ uncolored vertices.

Let C' be the coloring of T, which coincides with C_v on all vertices of T_v, except (perhaps) for v itself which is left uncolored, and coincides with C on all vertices outside T_v. Then C' contains either b_v or $b_v + 1$ black vertices in T_v. If v is black in C_v, then the number of black vertices in C' is equal to the number of black vertices in C, and the number of vertices left uncolored in C' is strictly smaller than that of vertices left uncolored by C, in contradiction to the optimality of C. Otherwise, C' contains $b_v + 1$ black vertices in T_v. By turning an arbitrary black vertex of T_v to be uncolored, we get that, again, the number of black vertices in C' is equal to that in C, and the number of vertices left uncolored in C' is strictly smaller than that of C, which leads to a contradiction. □

Algorithm 1 below constructs the array maxW, providing the maximal w for each value of b.

```
TreeBWC(T)
Input: A tree T with n vertices, rooted at r
Output: An array, providing the maximal W for each 0 ≤ b ≤ n

for each leaf v
   v.maxWhite[black]← [0] //the array for T_v if v is black
   v.maxWhite[white]← [1] //the array for T_v if v is white
   v.maxWhite[uncolored]← [0] //the array for T_v if v is uncolored
r.maxWhite ← generateTable(T)
for i ← 0 to n
   maxW[i] ← max{r.maxWhite[black][i],r.maxWhite[white][i],
                                        r.maxWhite[uncolored][i]}
return maxW
```

Algorithm 1: Finding the attributes of all BWCs of a tree

Algorithm 2 below separates the tree into smaller parts and works recursively on each of them. If the separator vertex is not the root of the tree, we deal with the subtree rooted at the separator and then continue dealing with the rest of the tree. Otherwise, we deal with the subtrees rooted at each child separately and finally merge their results. In general, after the tables for the roots of two subtrees T_1 and T_2, with a common root but otherwise disjoint sets of vertices, have been generated, the algorithm needs to generate the arrays for

u	A linked list of b-values
0	$b_1^0 \quad b_2^0$
1	NULL
2	$b_1^2 \quad b_2^2 \quad b_3^2 \quad b_4^2$
$\vdots$	$\vdots$
$\lg m_v + 3$	$b_1^{\lg m_v+3}$

Table 1: A table representing pairs (b, u)

$T' = T_1 \cup T_2$. This can be done simply by the following computations [5]. For any tree T, let B_T (W_T and U_T, resp.) be the array root(T).maxWhite[`black`] (root(T).maxWhite[`white`] and root(T).maxWhite[`uncolored`], resp.). We have:

$$\begin{array}{lcl} B_{T'}[b] & \leftarrow & \max\{B_{T_1}[b_1] + B_{T_2}[b_2] : b_1 + b_2 = b + 1\}, \\ W_{T'}[b] & \leftarrow & \max\{W_{T_1}[b_1] + W_{T_2}[b_2] - 1 : b_1 + b_2 = b\}, \\ U_{T'}[b] & \leftarrow & \max\{U_{T_1}[b_1] + U_{T_2}[b_2] : b_1 + b_2 = b\}. \end{array} \quad (1)$$

In fact, to reduce the runtime, we proceed somewhat differently. As a preparation to Algorithm 3, which performs these computations, Algorithm 2 below first translates all the values w in the table attached to each vertex v into the values $u = m_v - b - w$. By Lemma 2 we shall never need to use u-values exceeding $\lg m_v + 2$. Therefore, the algorithm converts the table attached to v into a $3 \times (\lg m_v + 3)$-table, whose entries are themselves lists, of varying lengths. The list at entry (c, u') of the table is composed of a sorted list of all b-values of the pairs with $u = u'$, assuming that v is colored c. (See, for example, Table 1.) This is done by Procedure SortU.

Algorithm 2 performs Algorithm 3 on pairs of lists, until it merges them all to a single one. In principle, we would like to merge at each step the two shortest lists; actually, we do not do it in this most economical way, but our ordering guarantees that we usually merge lists of average sizes. After each invocation of Algorithm 3 by Algorithm 2, the latter adjusts the resulting values, i.e., it subtracts 1 from all b-values in list$_i$[`black`] and from all the u-values in list$_i$[`uncolored`], where list$_i$ is the current table of lists built. The subtraction of 1 from the u-values in list$_i$[`uncolored`] is equivalent to a change of the index u of each list$_i$[`uncolored`][u] to $u - 1$. In order that after this adjustment we still keep all b-values for each $u \leq \lg m_v + 2$, Algorithm 3 computes these values for all $u \leq \lg m_v + 3$. After Algorithm 2 finishes dealing with the subtree rooted at v, it transforms all pairs in the resulting list back to their original form, which is a table that saves for each value of b and each color of v only the maximal w.

Example 3.1 *Consider the tree T_0 depicted in Figure 1. Table 2 shows the content of list$_1$ after each execution of the internal i-th iteration in Algorithm 2, when performed on the root of T_0. Here we show only the pairs (b, u) represented*

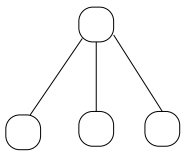

Figure 1: The tree T_0

iteration \ root color	black	white	uncolored
0	(2,0)	(0,0)	(1,1),(0,1)
1	(3,0)	(0,0)	(2,1),(1,1),(0,1)
2	(4,0)	(0,0)	(3,1),(2,1),(1,1),(0,1)

Table 2: The lists of pairs obtained by Alg. 2 on the root of T_0

by the lists, which are actually saved as shown in Table 1.

At the end of the process, after the algorithm has determined all pairs corresponding to optimal BWCs of a subtree T_v for some vertex v, it saves them in v.maxWhite and deletes all the children of v from the tree, along with their tables. This deletion is done by a procedure named reduce(T, v).

Algorithm 3 below gets two tables of lists representing pairs (b, u) and computes their pairwise sums by performing the procedure computePairwiseSums for each possible value of u. This procedure, performed with the parameters $\text{list}_1[q]$ and $\text{list}_2[p-q]$, computes the pairwise sums of the elements of its two parameters. Seidel (cf. [6]) gave a solution which takes $O(m \lg m)$ time, where m is the total size of the two lists. In fact, replacing each list by the generating polynomial of the numbers in the list, the set of pairwise sums is the set of powers for which the product of the two polynomial has a non-zero coefficient. Since two polynomials of degree m can be multiplied in time $O(m \lg m)$, the set of pairwise sums can be found also in time $O(m \lg m)$. For each sum (p, u), entry (p, u) of a temporary boolean table becomes true. At the end of the process, the resulting list contains all the pairs.

Recall that Algorithm 2 translates the pairs in each array from (b, w) to (b, u). In principle, we could have used the pairs (b, u) from the beginning of Algorithm 1. We prefer having the original kind of pairs (b, w) during the calculation, so that the values we obtain are more meaningful to the problem.

generateTable(T)

Input: A tree T rooted at r, with a $3 \times (m_v + 1)$-table attached to each leaf v

Output: The table r.maxWhite

$T' \leftarrow T$

if $|T'| \leq 2$

  solve the problem directly and return the resulting table

$s \leftarrow$ findSeparator(T')

if $s = r$

  $v_1, v_2, \ldots, v_d \leftarrow$ all children of s

  for $i = 1$ **to** d

    maxWhite$_i \leftarrow$ generateTable($T_{v_i} \cup \{s\}$)

    for $b = 0$ to m_{v_i}

      maxWhite$_i[b] \leftarrow m_{v_i} - b -$maxWhite$_i[b]$ //transform w's to u's

    list$_i \leftarrow$ SortU(maxWhite$_i$)

  for $k \leftarrow 0$ **to** $\lceil \lg d \rceil$

    for $i \leftarrow 1$ **to** $d - 2^{k+1} + 1$ **by** 2^{k+1}

      list$_i$[`black`]$\leftarrow$ Fusion(list$_i$[`black`],list$_{i+2^k}$[`black`])

      list$_i$[`white`]$\leftarrow$ Fusion(list$_i$[`white`],list$_{i+2^k}$[`white`])

      list$_i$[`uncolored`]$\leftarrow$ Fusion(list$_i$[`uncolored`],list$_{i+2^k}$[`uncolored`])

      adjust the b-vals in list$_i$[`black`] and the u-vals in list$_i$[`uncolored`]

  s.maxWhite $\leftarrow$ list$_1$ transformed back to its original form

if $s \neq r$

  s.maxWhite $\leftarrow$generateTable(T_s)

  $T' \leftarrow$ reduce(T', s) //delete all descendants of s to obtain a reduced tree

  r.maxWhite $\leftarrow$ generateTable(T')

return r.maxWhite

Algorithm 2: Solution of Problem 2

Input: A $3 \times (n+1)$-table maxWhite containing u-values

Output: A table containing $3 \times (\lg n + 2)$ lists of b-values, indexed by u

for $u \leftarrow 0$ to $\lg n + 1$

  for each $c \in \{\texttt{black}, \texttt{white}, \texttt{uncolored}\}$

    outlist$[c][u] \leftarrow$ empty list

for $b \leftarrow 0$ to n

  for each $c \in \{\texttt{black}, \texttt{white}, \texttt{uncolored}\}$

    $u \leftarrow$maxWhite$[c][b]$

    outlist$[c][u] \leftarrow$ outlist$[c][u] \cup b$

return outlist

Procedure: SortU(maxWhite)

3.3 Time Complexity

3.3.1 Runtime of Algorithm 3

The procedure computePairwiseSums(list$_1[q]$,list$_2[p-q]$) is performed exactly once for each p, q with $0 \leq q \leq p \leq \lg m_v + 3$, where m_v is the total size of the two lists. The runtime of this procedure, suggested by Seidel (cf. [6]), is $O(m_v \lg m_v)$. Therefore, the runtime of Algorithm 3 is $O(m_v \lg^3 m_v)$.

3.3.2 Runtime of Algorithm 2

Finding a separator vertex, at the beginning of the algorithm, takes $O(n)$ time. The transformation of pairs from the original form (b, w) in each maxWhite$_i$ to (b, u), and in the inverse direction in s.maxWhite, are done in $O(m)$ time.

Saving the arrays in a 2-dimensional table, sorted by the value of u, and for each u sorted by the value of b, is done by Procedure SortU in linear time in m.

The most time-consuming part of Algorithm 2 is the invocation of Algorithm 3.

Lemma 3 *For each vertex v having d_v children y_i, $1 \leq i \leq d_v$, attached with the tables y_i.maxWhite, Algorithm 2 finds the table v.maxWhite in $O(m_v \lg^3 m_v \lg d_v)$*

Fusion(list_1,list_2)

Input: Two $(\lg n + 3)$-size arrays of lists – list_1 and list_2 – containing u-values

Output: An array of lists containing all maximal pairwise sums of elements of list_1 and list_2

tmp $\leftarrow$ **new** boolean$[n+1][\lg n + 4]$ // All entries are initialized to `false`.
  // tmp$[b][u]$ =`true` iff $b_i \in \text{list}_i[u_i]$, $i = 1, 2$, where $\sum_i b_i = b$, $\sum_i u_i = u$.
for $p \leftarrow 0$ **to** $\lg n + 3$
  for $q \leftarrow 0$ **to** p
    for each $x \in$ computePairwiseSums($\text{list}_1[q]$,$\text{list}_2[p-q]$)
      tmp$[x][p] \leftarrow$ **true**
for $p \leftarrow 0$ **to** $\lg n + 3$
  for $i \leftarrow 0$ **to** n
    if $tmp[i][p]$
      outlist$[p] \leftarrow$ outlist$[p] \cup \{i\}$
return outlist

Algorithm 3: Merging subtrees rooted at siblings

time.

Proof: To find v.maxWhite, Algorithm 2 invokes Algorithm 3 on adjacent pairs of children. Therefore, denoting m_{y_i} by m_i, the runtime is bounded, up to a constant, by

$$\begin{aligned}
&\sum_{j=0}^{\lfloor d_v/2\rfloor-1}\left(\sum_{i=2j+1}^{2j+2} m_i\right)\lg^3\left(\sum_{i=2j+1}^{2j+2} m_i\right)+\sum_{j=0}^{\lfloor d_v/4\rfloor-1}\left(\sum_{i=4j+1}^{4j+4} m_i\right)\lg^3\left(\sum_{i=4j+1}^{4j+4} m_i\right)\\
&+\cdots+\left(\sum_{i=1}^{d_v} m_i\right)\lg^3\left(\sum_{i=1}^{d_v} m_i\right)\\
&\le \lg d_v\cdot\sum_{i=1}^{d_v} m_i\lg^3\left(\sum_{i=1}^{d_v} m_i\right)\\
&= m_v\lg^3 m_v\lg d_v.
\end{aligned}$$

□

In principle, the computations described in this lemma are the most time-consuming part of Algorithm 2, and therefore we could have estimated here the runtime of the algorithm. However, we prefer estimating the runtime in a way which is more suitable to the recursive nature of the algorithm.

The runtime R of Algorithm 2 on a tree $T_{n,m}$, with n vertices and m-size data, satisfies the recursion

$$R(T_{n,m})\le\begin{cases} R(T_{n_1,m})+R(T_{n-n_1+1,m})+c_1 n, & (2\le n_1<n),\\ & n>2,\ s\neq r,\\ \sum_{i=1}^{d} R(T_{n_i,m_i})+c_1 m+c_1 m\lg^3 m\lg d, & \\ & (\sum m_i=m,\sum n_i=n-1),\\ & n>2,\ s=r,\\ c_2 m, & n=2,\end{cases}$$

for some global constants $c_2\ge 2c_1\ge 2$, where the trees on the right-hand side of the formula are those attained by the separation, s is a separator vertex, r the root of $T_{n,m}$ and d the number of children of r. Recall that $m\ge n\ge 2$. For the case where $n>2$ and $s\neq r$, the algorithm first finds the separator in $O(n)$ time, then deals with the subtree rooted at s, turns s into a leaf, and then deals with the reduced tree $T_{n-|T_s|+1,m}$. For the case where $n>2$ and $s=r$, the algorithm first finds the separator in $O(n)$ time, then deals with all trees T_{n_i,m_i} rooted at the children of r, adds r as a new root to each T_{n_i,m_i} in $O(m_i)$ time, and finally performs Algorithm 3 to merge all subtrees. By Lemma 3, this takes $O(m\lg^3 m\lg d)$ time.

3.3.3 Conclusion of the Proof of Theorem 5

It suffices to prove that $R(T_{n,m})=O(nm\lg^3 m)$. We use induction on n. We actually prove that $R(T_{n,m})\le(c_2 n-2c_2)m\lg^3 m$. For $n=2$, the correctness

is trivial. Assume the inequality holds for $2 \leq n' < n$. If $s \neq r$, then

$$\begin{aligned} R(T_{n,m}) &\leq R(T_{n_1,m}) + R(T_{n-n_1+1,m}) + c_1 n \\ &\leq (c_2 \cdot n_1 - 2c_2)m \lg^3 m + (c_2(n - n_1 + 1) - 2c_2)m \lg^3 m + c_1 n \\ &= (c_2 n - 2c_2)m \lg^3 m - c_2 m \lg^3 m + c_1 n \\ &\leq (c_2 n - 2c_2)m \lg^3 m. \end{aligned}$$

If $s = r$, then $d \geq 2$ and

$$\begin{aligned} R(T_{n,m}) &\leq \textstyle\sum_{i=1}^{d} R(T_{n_i,m_i}) + c_1 m + c_1 m \lg^3 m \lg d \\ &\leq \textstyle\sum_{i=1}^{d} (c_2 n_i - 2c_2)m_i \lg^3 m_i + 2c_1 m \lg^3 m \lg d \\ &\leq \textstyle\sum_{i=1}^{d} (c_2 n_i - 2c_2)m \lg^3 m + 2c_1 m \lg^3 m \lg d \\ &= m \lg^3 m(c_2(n-1) - 2c_2 d + 2c_1 \lg d) \\ &\leq m \lg^3 m(c_2 n - 2c_2). \end{aligned}$$

This completes the proof of Theorem 5.

3.3.4 Conclusion of the Proof of Theorem 1

Algorithm 1 runs in linear time, except for the invocation of Algorithm 2. Therefore, its runtime is $O(nm \lg^3 m) = O(n^2 \lg^3 n)$.

4 Proof of Theorem 2

Given a tree T and two integers b, w, for which maxW$[b] \geq w$, we need to find a (b, w)-coloring of T.

To this end, Algorithm 2 needs to be changed so as to save a stack S_v for each vertex v. Each time the algorithm invokes Algorithm 3, it saves in the stack of the current separator the lists found by Algorithm 3. More explicitly, inside the inner i-loop of Algorithm 2 (after the adjustment of the b- and u-values), we add the line push(list$_i$, S_s), so that each output list of Algorithm 3 will be saved in the stack of the separator s. Here list$_i$ will be saved as an array of u-values, indexed by the values of b (see Example 4.1). In addition, the stack of each vertex v which is solved without being a separator (which happens, for example, if v is the root of the tree and has a single child), will contain the table of v, saved, again, as an array of u-values. Notice that, in Algorithm 2, except for the last result obtained for each separator vertex, each list$_i$, obtained by the invocation of Algorithm 3 in the k-th iteration also plays a role of an input list

black	(4,0)
white	(0,0)
uncolored	(3,1),(2,1),(1,1),(0,1)
black	(3,0)
white	(0,0)
uncolored	(2,1),(1,1),(0,1)

Table 3: The stack for the root of T_0

in the $(k+1)$-st iteration. Therefore, while trying to find a BWC for a vertex with a specific output list, we can simply consider the corresponding two input lists in the same stack. In order to do so, we simply save pointers from each list in the stack to the two lists in the stack it was obtained from. The pointers from the lists, saved in the bottom of the stack S_v, will be to the lists at the top of the stack S_u, where u is a child of v. While each v.maxWhite table is deleted when a subtree containing v is reduced to a leaf, the stack S_v is saved till the end of the algorithm.

Example 4.1 *Table 3 shows the stack for the root of* T_0*. Here, again, we show only the pairs* (b, u) *represented by the lists, which are actually saved as shown in Table 1. This stack contains only two elements. As we can see, the black list of the bottom of this stack contains the pair* $(3, 0)$*. This is due to the fact that this pair was obtained in the first iteration of Algorithm 2 on the root of* T_0 *(see Table 2). Since the pair* $(2, 1)$ *was obtained in the second (and last) iteration of Algorithm 2 on the root of* T_0*, the uncolored list of the top of this stack contains it (see Table 2).*

Algorithm 4, which finds the required coloring, invokes a new algorithm, `NewTreeBWC`, which is identical to Algorithm 1, except that it invokes the above updated version of Algorithm 2 instead of the original one. Later on, the algorithm translates its input from (b, w)-values to (b, u)-values, and eventually invokes Algorithm 5, which actually colors the tree. Algorithm 5, in turn, invokes the procedures ColorLeaf and ColorNode to handle the cases where a vertex is a leaf or has a single child, respectively.

As opposed to the runtime of Algorithm 2, which is not changed, the memory used is increased from $O(n)$ to $O(n^2)$. In fact, whereas in the original algorithm we kept data only for the leaves, in the updated algorithm we keep $O(dn)$-size

ColorTree(T, r, b, w)

Input: A tree T with a root r and two integers b and w

Output: A BWC of T with b black and w white vertices

NewTreeBWC(T)

if r.maxW$[b] < w$

there exists no BWC as required

else

$u \leftarrow |T| - b - w$

Color(T, r, b, u)

Algorithm 4: Main algorithm for finding a BWC

information for each vertex with degree d, and therefore of size $O(n^2)$ for the whole tree.

The runtime $T(n)$ of Algorithm 5, including the calls to ColorLeaf and ColorNode, satisfies $T(n) = T(a)+T(n-a+1)+n$, where $2 \leq a \leq n-1$ (a being the size of the left subtree of the root). By induction, we readily get $T(n) = O(n^2)$.

Thus, the total runtime of Algorithm 4 is basically the same as that of Algorithm 1, namely $O(n^2 \lg^3 n)$. This completes the proof.

5 Possible Improvements of Theorems 1 and 2

5.1 Proof of Proposition 3

1. A number b is economical if there exists a vertex v such that, when taking v as the root of the tree, there exist some subtrees, rooted at children of v, the sum of whose sizes is b. In this case, v separates the tree into two parts, of sizes b and $n - b - 1$. We record that b is economical and that v is a corresponding separator (which information is saved for the purposes of part 2). To find all economical b's, we can simply check all vertices of the tree. For each vertex v (regarded again as the root of the tree), with k children, say, we check all 2^k possible sums of the sizes of the subtrees rooted at its children. Each such sum is an economical b. Finding the sizes of all subtrees, combined over all the vertices of the tree, can be done in a linear time. For each vertex we find all the sums in time 2^k, where k is

Color(T, r, b, u)

Input: A tree T with a root r and two integers b and u

Output: A BWC of T with b black and $|T| - b - u$ white vertices

$v_1, \ldots, v_d \leftarrow$ all children of r

if $d = 0$

  ColorLeaf(r, b, u)

if $d = 1$

  ColorNode(T, r, b, u)

else

  uArr $\leftarrow$ pop(S_r)

  for $c \leftarrow$ `black` **to** `uncolored`

    if uArr$[c][b] \leq u$ // This condition will be satisfied for at least one c

      r.color$\leftarrow c$

      if $c =$ `uncolored`

        $u \leftarrow u + 1$

      if $c =$ `black`

        $b \leftarrow b + 1$

      for $i \leftarrow 0$ **to** b

        if uArr.pointer2$[c][i]$+uArr.pointer1$[c][b-i] \leq u$

          Color$(T_{v_1} \cup \cdots \cup T_{v_{\lceil \frac{d}{2} \rceil}} \cup \{r\}, r, b-i$,uArr.pointer1$[c][b-i])$

          Color$(T_{v_{\lceil \frac{d}{2} \rceil + 1}} \cup \cdots \cup T_{v_d} \cup \{r\}, r, i$,uArr.pointer2$[c][i])$

          return

Algorithm 5: Finding a BWC as required

Input: A vertex r and two numbers b, u

Output: A coloring of r

if $b = 1$

  r.color $\leftarrow$ black

else if $u = 1$

  r.color $\leftarrow$ uncolored

else

  r.color $\leftarrow$ white

Procedure: ColorLeaf(r, b, u)

Input: A tree T rooted at r, having one child, and two numbers b, u

Output: A coloring of T

$v_1 \leftarrow$ the child of r

if $u = r$.maxWhite[`black`][b]

  r.color $\leftarrow$ black

  $b \leftarrow b - 1$

else if $u = r$.maxWhite[`white`][b]

  r.color $\leftarrow$ white

else if $u = r$.maxWhite[`uncolored`][b]

  r.color $\leftarrow$ uncolored

  $u \leftarrow u - 1$

Color(T_{v_1}, v_1, b, u)

Procedure: ColorNode(T, r, b, u)

the degree of the vertex. Thus, as long as the maximal degree in the tree is $o(\lg n)$, the runtime for each vertex is at most $2^{o(\lg n)} = n^{o(1)}$. By [8], in almost all trees the maximal degree is $\frac{c \lg n}{\lg \lg n}$ for some constant c. Therefore, in almost all trees the total runtime is at most $n \cdot 2^{\frac{c \lg n}{\lg \lg n}} = n^{1+o(1)}$.

2. Given an economical b, the separator vertex $v = v(b)$, found in the previous part, will be the only uncolored vertex in the tree. Now we only have to check all possible sums of sizes of the subtrees rooted at the children of v, in order to find a collection which adds up to b. All these subtrees should be colored black. The rest of the vertices in the tree are colored white. Therefore, after we know all the economical b's, in order to find a BWC for an economical b, we know in $n^{o(1)}$ time which vertex should be colored in each color. To actually color each vertex, we need linear time.

5.2 The Statistics of Economical b's

We first tested the average percentage of economical b's for trees of sizes $n = 50, 100, \ldots, 1000$. For each such size n, we selected 1000 random n-vertex trees, and found the average percentage of economical b's. (These random labeled trees were generated using Prüfer code [9].) The results are depicted in Figure 2. We see that the percentage of economical b's decreases slowly as a function of n.

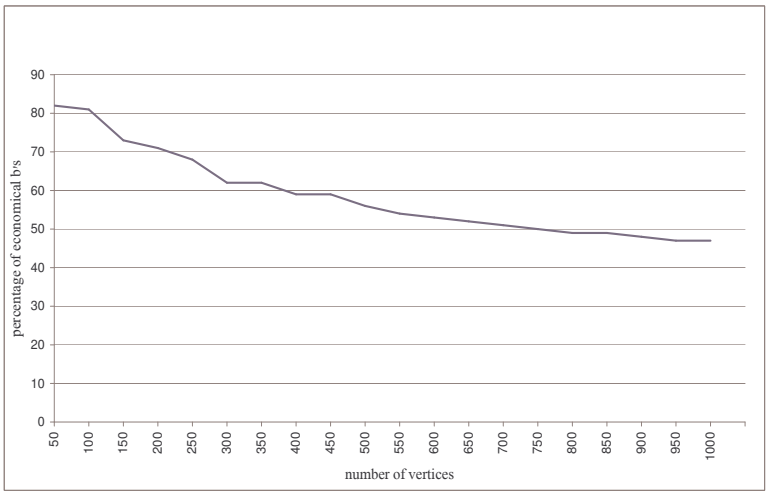

Figure 2: Percentage of economical b's in trees of various sizes

Figure 3 shows the probability of each value of b to be economical. In order to find it, we tested 1000 random trees with 10000 vertices and, for each $b \in [1, 9999]$, found its probability of being economical. To avoid random jumps, we actually recorded for each $x = 250, 750, \ldots, 9750$ the average value for all b's in the interval $[x - 249, x + 250]$ (excluding $b = 10000$). The symmetry of the graph with respect to the line $b = 5000$ is due to the fact that a value b is economical if and only if $n - b - 1$ is such.

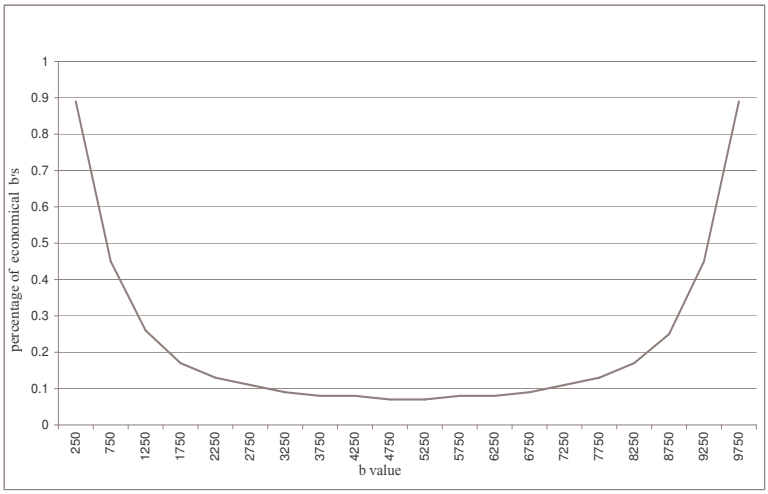

Figure 3: The probability of each b to be economical

6 Conclusion

Given a tree T, all pairs (b, w) representing optimal BWCs can be found by Algorithm 1 in time $O(n^2 \lg^3 n)$. The coloring for a specific pair can be then found in $O(n^2)$ time. Finding an optimal BWC for an economical b is done in $n^{1+o(1)}$ time.

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, E. Korach and S. Zucker, Anticoloring and separation of graphs, preprint.

[3] N. Bray, from MathWorld–A Wolfram web resource, created by E. W. Weisstein. *url:* mathworld.wolfram.com/GraphStrongProduct.html

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

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

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

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

[8] J. W. Moon, On the maximum degree in a random tree, *Michigan Math. J.*, 15/4:429–432, 1968.

[9] Prüfer, H., Neuer beweis eines satzes über permutationen, *Arch. Math. Phys.*, 27:742–744, 1918.

[10] E. W. Weisstein, from MathWorld–A Wolfram web resource. *url:* mathworld.wolfram.com/CartesianProduct.html

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

[12] B. Zelinka, Medians and peripherians of trees, *Arch. Math.*, 4:87–95, 1968.