Improved Fixed-Parameter Algorithms for Minimum-Flip Consensus Trees

SEBASTIAN BÖCKER and QUANG BAO ANH BUI and ANKE TRUSS

Lehrstuhl für Bioinformatik, Friedrich-Schiller-Universität Jena,
Ernst-Abbe-Platz 2, 07743 Jena, Germany
sebastian.boecker,quangbaoanh.bui,anke.truss@uni-jena.de

Abstract. In computational phylogenetics, the problem of constructing a consensus tree for a given set of rooted input trees has frequently been addressed. In this paper we study the MINIMUM-FLIP PROBLEM: the input trees are transformed into a binary matrix, and we want to find a perfect phylogeny for this matrix using a minimum number of flips, that is, corrections of single entries in the matrix. The graph-theoretical formulation of the problem is as follows: Given a bipartite graph $G = (V_t \cup V_c, E)$, the task is to find a minimum set of edge modifications such that the resulting graph has no induced path with four edges that starts and ends in V_t, where V_t corresponds to the taxa set and V_c corresponds to the character set.
We present two fixed-parameter algorithms for the MINIMUM-FLIP PROBLEM, one with running time $O(4.83^k + poly(m, n))$ and another one with running time $O(4.42^k + poly(m, n))$ for n taxa, m characters, k flips, and $poly(m, n)$ denotes a polynomial function in m and n. Additionally, we discuss several heuristic improvements. We also report computational results on phylogenetic data.[1]

1 Introduction

When studying the relationship and ancestry of current organisms, discovered relations are usually represented as phylogenetic trees, that is, rooted trees where each leaf corresponds to a group of organisms, called *taxon*, and inner vertices represent hypothetical last common ancestors of the organisms located at the leaves of its subtree.

Supertree methods assemble phylogenetic trees with shared but overlapping taxon sets into a larger supertree that contains all taxa of every input tree and describes the evolutionary relationship of these taxa [2]. Constructing a supertree is easy for compatible input trees [3, 14], that is, in case there is no contradictory information encoded in the input trees. The major problem of supertree methods is dealing with incompatible data in a reasonable way [17]. The most popular supertree method is matrix representation with parsimony (MRP) [2]: MRP performs a maximum parsimony analysis on a binary matrix representation of the set of input trees. The Maximum-Parsimony Problem is NP-complete [7], and so is MRP. The matrix representation with flipping (MRF) supertree method also uses a binary matrix representation of the *rooted* input trees [6]. Unlike MRP, MRF seeks the minimum number of "flips" (corrections) in the binary matrix that make the matrix representation consistent with a phylogenetic tree. Evaluations by Chen et al. [5] indicate that MRF is superior to MRP and other common approaches for supertree construction, such as MinCut supertrees [17].

If all input trees share the same set of taxa, the supertree is called a consensus tree [1, 12]. As for supertrees, we can encode the input trees in a binary matrix. Ideally, the input trees match nicely and a consensus tree can be constructed without changing the relations

[1] This is a preprint of: Sebastian Böcker, Quang Bao Anh Bui and Anke Truss. Improved Fixed-Parameter Algorithms for Minimum-Flip Consensus Trees. *ACM T. Algorithms*, 8(1):17 pages, 2012. Article 7.

between taxa. In this case, we can construct the corresponding *perfect phylogeny* in $O(mn)$ time for n taxa and m characters [11]. Again, the more challenging problem is how to deal with incompatible input trees. Many methods for constructing consensus trees have been established, such as majority consensus or Adams consensus [1]. One method for constructing consensus trees is the *Minimum-flip* method [6]: Flip as few entries as possible in the binary matrix representation of the input trees such that the matrix admits a perfect phylogeny. Unfortunately, the problem of finding the minimum set of flips that make a matrix compatible (MINIMUM-FLIP PROBLEM), is NP-hard [6]. Based on a graph-theoretical interpretation of the problem and the forbidden subgraph paradigm of Cai [4], Chen et al. [6] introduce a simple fixed-parameter algorithm with running time $O(6^k\,mn)$, where k is the minimum number of flips. Recently, Komusiewicz and Uhlmann [13] introduced a kernel with $O(k^3)$ vertices for this problem. Furthermore, the problem can be approximated with approximation ratio $2d$ where d is the maximum number of ones in a column [6].

Our contributions We introduce two fixed-parameter algorithms for the MINIMUM-FLIP PROBLEM with running time $O(4.83^k + poly(m,n))$ and $O(4.42^k + poly(m,n))$ where $poly(m,n)$ denotes a polynomial function in m and n. We also discuss some heuristic improvements to reduce the running time of our algorithms in practice. Furthermore, we implemented our 4.83^k-algorithm and the fixed-parameter algorithm of Chen et al. [6] and applied them to perturbed matrix representations of phylogenetic trees, to evaluate the performance of our algorithm and to compare it to the algorithm of Chen et al.. Our 4.83^k-algorithm turns out to be significantly faster than the $O(6^k\,mn)$ strategy, and also much faster than worst-case running times suggest. We believe that our work is a first step towards exact computation of minimum-flip supertrees.

2 Preliminaries

A *parameterized problem* consists of an input pair (I,k) where I is the problem instance and k is the parameter. An algorithm that solves a parameterized problem in time $f(k)n^{O(1)}$ is called a *fixed-parameter algorithm*. A parameterized problem is *fixed-parameter tractable* if there is a fixed-parameter algorithm for the problem. See [8, 15] for more information.

Let $\mathcal{A}$ be a search tree algorithm. If $\mathcal{A}$ solves a size n problem instance by calling itself recursively for problem instances of size $n-d_1$, $n-d_2$, …, $n-d_i$ for a constant i, then $(d_1, d_2, \ldots, d_i)$ is called the *branching vector* of this recursion. The branching vector $(d_1, d_2, \ldots, d_i)$ corresponds to the running time recurrence: $T_n = T_{n-d_1} + T_{n-d_2} + \cdots + T_{n-d_i}$. Using the characteristic polynomial, we can compute a *branching number* α from this branching vector, such that the resulting search tree has size $O(\alpha^n)$. See [15] for details.

From now on, let n be the number of taxa and m be the number of characters. Let M be an $n \times m$ binary matrix that represents the characteristics of our taxa: Each cell $M[i,j]$ takes a value of '1' if the taxon t_i has character c_j, and '0' otherwise.

We say that M admits a *perfect phylogeny* if there is a rooted tree such that each of the n leaves corresponds to one of the n taxa and, for each character c_j, there is an inner vertex of the tree such that for all taxa t_i, $M[i,j] = 1$ if and only if t_i is a leaf of the subtree below c_j. The PERFECT PHYLOGENY problem is to recognize if a given binary matrix M admits a perfect phylogeny. Gusfield [11] introduced an algorithm that checks if a matrix M admits a perfect phylogeny and, if possible, constructs the corresponding phylogenetic tree in running time $O(mn)$.

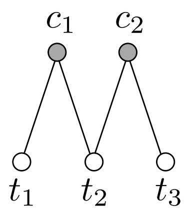


Fig. 1. An M-graph. Grey vertices denote characters, white vertices denote taxa.

The Minimum-Flip Problem [6] asks for the minimum number of matrix entries to be flipped from '0' to '1' or from '1' to '0' in order to transform M into a matrix that admits a perfect phylogeny. The corresponding decision problem is to check whether there exists a solution with at most k flips. Note that this decision problem is a parameterized problem with the parameter k. Our fixed-parameter algorithm requires a maximum number of flips k to be known in advance: To find an optimal solution we call this algorithm repeatedly, increasing k.

In this article, we use a graph-theoretical model to analyze the Minimum-Flip Problem. First, we define a graph model of the binary matrix representation. The *character graph* $G = (V_t \cup V_c, E)$ of an $n \times m$ binary matrix M is an undirected and unweighted bipartite graph with $n + m$ vertices $t_1, \ldots, t_n, c_1, \ldots, c_m$ where $\{c_i, t_j\} \in E$ if and only if $M[i, j] = 1$. The vertices in V_c represent characters and those in V_t represent taxa. We call the vertices c- or t-vertices, respectively.

An *M-graph* is a path of length four where the end vertices and the center vertex are t-vertices and the remaining two vertices are c-vertices, see Fig. 1. We call a graph *M-free* if is does not have an M-graph as an induced subgraph. The following theorem of Chen et al. [6] provides an essential characterization with regard to the graph-theoretical modeling of the Minimum-Flip Problem.

Theorem 1. *A binary matrix M admits a perfect phylogeny if and only if the corresponding character graph G does not contain an induced M-graph.*

With Theorem 1 the Minimum-Flip Problem is equivalent to the following graph-theoretical problem: Find a minimum set of edge modifications, that is, edge deletions and edge insertions that transform the character graph of the input matrix into an M-free bipartite graph. Using this characterization, Chen et al. [6] introduce a simple fixed-parameter algorithm with running time $O(6^k\, mn)$ where k is the minimum number of flips. This algorithm follows a search-tree technique from [4]: It identifies an M-graph in the character graph and branches into all six possibilities of deleting or inserting one edge of the character graph such that the M-graph is eliminated (four cases of deleting one existing edge and two cases of adding a new edge).

The following notation will be used frequently throughout this article: Let $N(v)$ be the set of neighbors of a vertex v. For two c-vertices $c_i, c_j \in V_c$, let $X(c_i, c_j) := N(c_i) \setminus N(c_j)$, $Y(c_i, c_j) := N(c_i) \cap N(c_j)$, and $Z(c_i, c_j) := N(c_j) \setminus N(c_i)$, see Fig. 2. We call c_i and c_j *c-neighbors* if and only if $Y(c_i, c_j)$ is not empty. Furthermore, an edge modification is *associated* with a vertex v (v can be a c-vertex or a t-vertex), if it inserts or deletes an edge incident to v.

3 Data reduction

Given a character graph G as input, we use the data reduction rules introduced in this section to cut down the size of G. In our search tree algorithms, these data reduction rules are also executed in the beginning of each recursive call. Reduction Rules 1 and 2 are fairly simple:

Rule 1: Delete all c-vertices $v \in V_c$ of degree $|V_t|$ from the graph.

Rule 2: Delete all c-vertices $v \in V_c$ of degree one from the graph.

Lemma 1. *Rule 1 is correct.*

Proof. Let c be a c-vertex of degree $|V_t|$ in the input graph G. Then, c is connected to all t-vertices in G and there cannot be an M-graph containing c. Furthermore, it is not possible to insert any new edge incident to c. Assume there is an optimal solution for G that deletes edges incident to c in G. If we execute all edge modifications of the optimal solution except deletions of edges incident to c, we also obtain an M-free graph, since every M-graph that does not contain c is destroyed by edge modifications of the optimal solution and there is no M-graph containing c. This is a contradiction to the assumption that the solution is optimal, so edges incident to c-vertices of degree $|V_t|$ are never deleted in an optimal solution. Thus, the corresponding c-vertices need not be observed and can be removed safely. □

The correctness of Rule 2 is obvious.

After computing and saving the degree of every vertex in G in time $O(mn)$, Rules 1 and 2 of the data reduction can be done in time $O(m+n)$.

Rule 3: For every c-vertex c of degree two, set $\{c,t\}$ as *forbidden* for all $t \in V_t \setminus N(c)$, i.e., in later stages of our algorithms, we will not insert any edge incident to c.

Lemma 2. *Rule 3 is correct.*

Proof. Let G be a character graph and c be a c-vertex of degree two in G. Let G' be the graph resulting from an optimal set of edge modifications on G. Assume that G' owns an edge (c,t) not contained in G.

We will now show that there is another optimal solution for G without inserting (c,t). Let t_i, t_j be the t-vertices connected with c in G. Since all M-graphs eliminated by inserting $\{c,t\}$ into G contain edges $\{c,t_i\}$ and $\{c,t_j\}$, we can delete $\{c,t_i\}$ or $\{c,t_j\}$ from G to eliminate these M-graphs instead of adding $\{c,t\}$ to G. Deleting an edge can only cause new M-graphs containing the c-vertex incident to this edge. But after the removal of one of the edges $\{c,t_i\}$ or $\{c,t_j\}$, c has degree one and cannot be contained in any M-graph. Therefore, the resulting graph is still M-free and the number of edge modifications does not increase since we swap an insertion for only one deletion. Hence, we can prohibit inserting edges incident to degree-two c-vertices without changing the cost of the optimal solution. □

Rule 3 can be easily done in time $O(m)$.

Corollary 1. *If every c-vertex in a character graph has degree two, there is an optimal solution for the* MINIMUM-FLIP PROBLEM *without inserting any edge into the character graph.*

Rule 4 of our data reduction is based on the following lemma.

Lemma 3. *Let $V_c' \subseteq V_c$ be a set of c-vertices with the same neighborhood. If there is an optimal solution that executes edge modifications associated with a c-vertex in V_c', there is an optimal solution that executes the same edge modifications on all c-vertices in V_c'.*

Proof. Let c_1, c_2 be two arbitrary c-vertices in V_c'. Since c_1 and c_2 share the same neighborhood, there is no M-graph containing only c_1 and c_2. Assume that the length-four path (t_1, c_1, t_2, c, t) with $t_1, t_2 \in N(c_1)$, $c \in V_c \setminus V_c'$ and $t \in V_t \setminus N(c_1)$ forms an M-graph. The length-four path (t_1, c_2, t_2, c, t) is also an M-graph. To eliminate these M-graphs, we have to execute suitable edge modifications.

Case 1: If the optimal solution comprises edge modifications that are associated with c, then no edge modifications associated with c_1 or c_2 are necessary to eliminate M-graphs (t_1, c_1, t_2, c, t) and (t_1, c_2, t_2, c, t).

Case 2: Assume that the optimal solution comprises different edge modifications associated with c_1 and c_2 to eliminate the M-graphs (t_1, c_1, t_2, c, t) and (t_1, c_2, t_2, c, t) . Without loss of generality, the optimal solution inserts an edge connecting c_1 with t and deletes at least an edge incident to c_2. The edge deletions associated with c_2 can be replaced by inserting the edge $\{c_2, t\}$ without creating a new M-graph containing c_1 and c_2, because c_1 and c_2 have the same neighborhood. This modification does not increase the cost of the optimal solution in this step.

Thus, there is an optimal solution that executes analogous edge modifications for all vertices in V_c. □

As a consequence of the above lemma, whenever we execute an edge modification associated with a c-vertex in a set of c-vertices with the same neighborhood, we execute the analogous edge modification on all other c-vertices in this set. To make use of this observation, we introduce the following *merge operation.*

Merge operation Let $V_c' \subsetneq V_c$ be a set of c-vertices that have the same neighborhood. The merge operation joins every c-vertex in V_c' into a new c-vertex c and connects c with all t-vertices in the neighborhood of V_c' and sets the cost of deleting or inserting an edge incident to c to $|V_c'|$.

The merge operation ensures that every edge modification associated with a c-vertex in a set of c-vertices with the same neighborhood will be executed to all c-vertices in this set, and the corresponding edit costs are correctly estimated.

Rule 4: Merge c-vertices with the same neighborhood.

The correctness of Rule 4 can be derived directly from Lemma 3.

In Sec. 4, we describe an algorithm that solves the MINIMUM-FLIP PROBLEM in time $O(4.83^k + poly(m, n))$. In Sec. 5, we introduce an improved version of our algorithm with running time $O(4.42^k + poly(m, n))$. Note that neither algorithm can be used to solve arbitrary *weighted* MINIMUM-FLIP PROBLEM instances where edit costs are different from edge to edge. However, our algorithms can be used to solve instances, where weights are assigned to c-vertices, i.e., edge modifications associated with one c-vertex must have identical weight.

4 The $O(4.83^k + poly(m, n))$ algorithm

In this algorithm we distinguish two cases: (i) the character graph G contains c-vertices of degree at least three, and (ii) every c-vertex of G has degree two. Note that every c-vertex

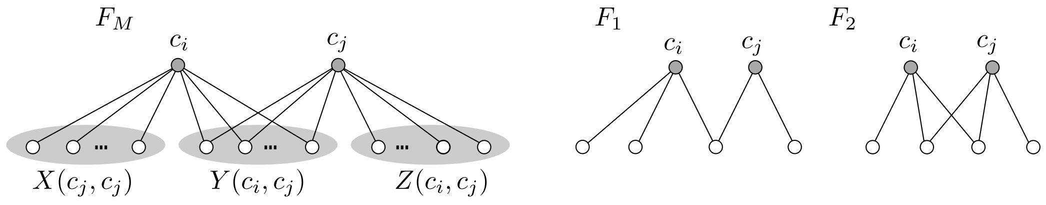


Fig. 2. The big M-graph F_{M} and its special cases F_1 and F_2.

of degree one is removed by our data reduction. In Case (i), we use the branching strategy described in Sec. 4.1. In Sec. 4.2, we show that in Case (ii) the problem can be solved in polynomial time, but to preserve the worst-case running time of our algorithm, we use a special branching strategy described in the same section.

4.1 Solving instances with c-vertices of degree at least three

In this section, we describe the branching strategy we use as long as there are c-vertices of degree three or higher. The efficiency of this branching is based on the following observation:

Lemma 4. *A character graph G reduced with respect to the abovementioned data reduction rules has a c-vertex of degree at least three if and only if G contains F_1 or F_2 (see Fig. 2) as an induced subgraph.*

Proof. Assume that there are c-vertices in G with degree at least three. Let c_i be a c-vertex with maximum degree in G. Then c_i must have a c-neighbor c_j, which has at least one neighbor t_j outside $N(c_i)$ because otherwise c_i would be removed from G by the data reduction. Let t_k be a common neighbor of c_i and c_j that has to exist since c_i and c_j are c-neighbors. There also exists a neighbor t_i of c_i that is not a neighbor of c_j. If t_i is the only t-vertex that is a neighbor of c_i but not of c_j, then c_i and c_j must share another common neighbor t'_k besides t_k (since t_i has degree at least three) and the M-graph $t_i c_i t_k c_j t_j$ and edges $\{c_i, t'_k\}, \{c_j, t'_k\}$ form an F_2 graph. Otherwise let t'_i be an another t-vertex, which is a neighbor of c_i but not of c_j, the M-graph $t_i c_i t_k c_j t_j$ and the edge $\{t'_i, c_i\}$ form an F_1 graph. We conclude that if G has t-vertices with degree at least three, then G contains F_1 or F_2 as induced subgraph.

If G contains F_1 or F_2 as induced subgraph, it is obvious that G has c-vertices of degree at least three. □

We now consider the following structure of intersecting M-graphs, called *big M-graph*: this graph is a subgraph of the character graph and consists of two c-vertices c_i, c_j and t-vertices in the nonempty sets $X(c_i, c_j)$, $Y(c_i, c_j)$, $Z(c_i, c_j)$ where at least one of these sets has to contain two or more t-vertices, see Fig. 2. The graphs F_1 and F_2 are big M-graphs of minimum size. In view of Lemma 4, any character graph with at least one c-vertex of degree three or higher has to contain big M-graphs as induced subgraphs. Furthermore, it should be clear that a big M-graph contains many M-graphs as induced subgraphs. Therefore, if there are big M-graphs in the character graph, our algorithm first branches into subcases to eliminate all M-graphs contained in those big M-graphs. The branching strategy to eliminate all M-graphs contained in a big M-graph is based on the following lemma:

Lemma 5. *If a character graph G is M-free, then for every two distinct c-vertices c_i, c_j of G it holds that at least one of the sets $X(c_i, c_j)$, $Y(c_i, c_j)$, $Z(c_i, c_j)$ must be empty.*

Since there is at least one M-graph containing c_i and c_j if $X(c_i, c_j)$, $Y(c_i, c_j)$, $Z(c_i, c_j)$ are simultaneously non-empty, the correctness of Lemma 5 is obvious.

Lemma 5 leads to the following branching strategy for big M-graphs. Given a character graph G with at least one c-vertex of degree three or higher, our algorithm chooses a big M-graph F_{M} in G and branches into subcases to eliminate all M-graphs in F_{M}. Let c_i, c_j be the c-vertices of F_{M}. According to Lemma 5, one of the sets $X(c_i, c_j)$, $Y(c_i, c_j)$, $Z(c_i, c_j)$ must be "emptied" in each subcase. Let x, y, z denote the cardinalities of sets $X(c_i, c_j), Y(c_i, c_j)$ and $Z(c_i, c_j)$. We now describe how to empty $X(c_i, c_j)$, $Y(c_i, c_j)$, and $Z(c_i, c_j)$.

For each t-vertex t in $X(c_i, c_j)$ there are two possibilities to remove it from $X(c_i, c_j)$: we either disconnect t from the big M-graph by deleting the edge $\{c_i, t\}$, or move t to $Y(c_i, c_j)$ by inserting the edge $\{c_j, t\}$. Therefore, the algorithm has to branch into 2^x subcases to empty $X(c_i, c_j)$ and in each subcase, it executes x edge modifications. The set $Z(c_i, c_j)$ is emptied analogously.

To empty the set $Y(c_i, c_j)$ there are also two possibilities for each t-vertex t in $Y(c_i, c_j)$, namely moving it to $X(c_i, c_j)$ by deleting the edge $\{c_j, t\}$ or moving it to $Z(c_i, c_j)$ by deleting the edge $\{c_i, t\}$. This also leads to 2^y subcases and in each subcase, y edge modifications are executed.

Altogether, the algorithm branches into $2^x + 2^y + 2^z$ subcases when dealing with a big M-graph F_{M}. In view of Lemma 5, at least $\min\{x, y, z\}$ edge modifications must be executed to eliminate all M-graphs in F_{M}. In the worst case, each edge modification has weight 1 and our branching strategy has branching vector

$$(\underbrace{x, \ldots, x}_{2^x}, \underbrace{y, \ldots, y}_{2^y}, \underbrace{z, \ldots, z}_{2^z}). \tag{1}$$

This leads to a branching number of 4.83 as shown in the following lemma (see [15] for details on branching vectors and branching numbers).

Lemma 6. *The worst-case branching number of the above branching strategy is* 4.83.

Proof. The branching number b of the above branching strategy is the single positive root of the equation

$$2^x \frac{1}{b^x} + 2^y \frac{1}{b^y} + 2^z \frac{1}{b^z} = 1 \iff \frac{1}{(b/2)^x} + \frac{1}{(b/2)^y} + \frac{1}{(b/2)^z} = 1.$$

If we consider $\frac{b}{2}$ a variable, the single positive root of the second equation is the branching number corresponding to the branching vector (x, y, z). The smaller values x, y, z take, the higher $\frac{b}{2}$ and, hence, b. Due to the definition of a big M-graph, x, y, and z cannot equal one simultaneously, so b is maximal if one of the variables x, y, z equals two and the other two equal one. Without loss of generality, assume that $x = 2$ and $y = z = 1$. Then the single positive root of the equation $(\frac{2}{b})^2 + \frac{2}{b} + \frac{2}{b} = 1$ is $\frac{b}{2} = 2.414214$. Therefore, b is at most 4.83. $\square$

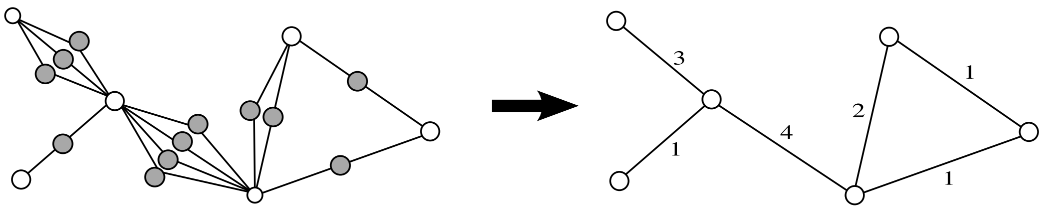


Fig. 3. Left: When all c-vertices (gray) have degree two, we can regard each c-vertex with its incident edges as a single edge in a multigraph whose vertex set is the set of t-vertices (white). Right: We merge all those "edges" between two t-vertices into a single weighted edge whose weight equals the number of c-vertices adjacent to the t-vertices, and we obtain a simplified model of our graph.

From Lemma 5, we infer an interesting property. If we consider that we need an edge modification for each vertex we remove from set X, Y, or Z, the following corollary is a straightforward observation.

Corollary 2. *There is no solution with at most k flips if there exist two c-vertices $c_i, c_j \in V_c$ satisfying* $\min\{|X(c_i, c_j)|, |Y(c_i, c_j)|, |Z(c_i, c_j)|\} > k$.

We use this property for pruning the search tree in the implementation of our algorithm, see Sec. 6.

Corollary 2 implies that we can abort a program call whenever we find a big M-graph where x, y, z simultaneously exceed k. Furthermore, if one or two of the values x, y, z are greater than k, we do not branch into subcases deleting the respective sets. Anyway, the number of subroutine calls in this step of the algorithm is fairly large, up to $3 \cdot 2^k$. But large numbers of program calls at this point are a result of large numbers of simultaneous edge modifications, which lower k to a greater extent. For simplicity, assume that $x = y = z = k$, we see that our algorithm branches in $3 \cdot 2^k$ subcases, but at the same time it consumes all available flips. Therefore, the depth of the corresponding search tree is 1 and its size is $3 \cdot 2^k$. Thus, the branching number of our branching strategy goes to 2 for large x, y, z and this is confirmed by the growth of running times in our computational experiments (see Sec. 7).

4.2 Solving instances with c-vertices of degree at most two

In this section, we assume that there is no big M-graph in the character graph. As we proved in Lemma 4, if a character graph G reduced with respect to our data reduction does not contain any big M-graph, then every c-vertex in G has degree two. According to Corollary 1, we never have to insert edges into this graph G.

Now that all c-vertices in our graph G have degree two, we define a weighted graph G_w as follows: We adopt the set V_t of t-vertices in G as vertex set for G_w. Two vertices t_1, t_2 are connected if and only if they possess a common neighbor in G. The weight of an edge $\{t_1, t_2\}$ is the total weight of the edges connecting t_1 (or t_2) with the common neighbors of t_1 and t_2 in the bipartite graph G, see Fig. 3.

One can easily see that there is an M-graph in G if and only if there is a path of length two in G_w. On the weighted graph G_w, the Minimum-flip Problem turns out to be the problem of deleting a set of edges with minimum total weight such that there are no paths of length two in G_w, that is, the graph is split into connected components of size one or two. In other words, the remaining graph is a matching. Moreover, one can easily see that the larger the weight of the matching, the smaller the weight of the deleted edges. Therefore, the

Minimum-flip Problem becomes the problem of finding a *maximum weighted matching*. Furthermore, one can easily recognize that the number of edges in G_w is bounded by m, the number of character vertices. Gabow [9] introduced an algorithm to compute the maximum weighted matching of a graph (V, E) in time $O(|V||E| + |V|^2 \log(|V|))$. Since the number of edges in G_w is bounded by m, using Gabow's algorithm [9], we can solve the problem in time $O(n \cdot m + n^2 \log(|V|))$ if every c-vertex is of degree at most two.

Clearly, it is advisable to use Gabow's algorithm in practice. Unfortunately, this would lead to an increased polynomial part in the worst-case running time of our algorithm claimed in Theorem 2. It turns out that we can also handle such instances efficiently using a simple branching strategy. Solely for this theoretical purpose, we now describe this branching strategy.

Deleting a weighted edge in G_w corresponds to deleting one of the edges incident to each of the respective c-vertices in G. That is, whenever we delete an edge $\{t_i, t_j\}$ of weight m from G_w, we include, for each vertex c of the m original c-vertices that were used for e (see Fig. 3), one of the edges $\{c, t_i\}$ and $\{c, t_j\}$ from G in our solution set and lower k by m.

We now describe our branching strategy for the weighted problem in detail. If we consider an edge e in G_w, we observe that either e has to be deleted or all other edges that are incident to a vertex in e.

We pick an edge e that has weight greater than one, shares a vertex with an edge of weight greater than one, or is incident to a vertex of degree three or higher. Then, we branch into two cases: Delete e or keep e but delete all edges that share vertices with e. In each case, lower k by the weights of the deleted edges. If the graph decomposes, we treat each connected component separately. With this branching strategy we receive a branching vector of $(1, 2)$ or better, which corresponds to a branching number of 1.62. We use this strategy as long as there are degree-three vertices or edges of weight greater than one.

As soon as all edges have weight one and all vertices have degree at most two, the remaining graph is either a path or a cycle. We solve each of these graphs by alternately keeping and deleting edges such that solving a path with l edges costs $\lfloor \frac{l}{2} \rfloor$ and solving a cycle of length l costs $\lceil \frac{l}{2} \rceil$. Clearly, this operation can be done in linear time. This finishes our branching strategy for the Minimum-Flip Problem when every c-vertex is of degree two.

We can now show that our algorithm solves the Minimum-Flip Problem in time $O(4.83^k(m + n) + mn)$.

Theorem 2. *The above algorithm solves the* Minimum-Flip Problem *for a character graph with n t-vertices and m c-vertices in $O(4.83^k\,(m + n) + mn)$ time.*

Proof. At the beginning of the algorithm, the execution of the data reduction takes $O(mn)$ time. By saving the degree of each vertex in G, the algorithm needs $O(m + n)$ time to execute the data reduction in each recursion call.

The algorithm distinguishes two cases: there are c-vertices with degree at least three, or every c-vertex has degree two. In the first case, it uses the branching strategy described in Sec. 4.1 with branching number 4.83, in the second case it executes the branching strategy in Sec. 4.2 with branching number 1.62. Therefore, the size of the search tree is $O(4.83^k)$. Therefore, the running time of the algorithm is $O(4.83^k\,(m + n) + mn)$. □

Using Gabow's algorithm if every c-vertex is of degree at most two would increase the running time of the algorithm to $O(4.83^k\,(m + n) + mn + n^2 \log(n))$.

We can use interleaving [16] by repeatedly applying the polynomial kernelization from [13] during the course of the search tree algorithm. This improves the worst-case running time of our algorithm to $O(4.83^k + poly(m, n))$ as claimed. Note that the hidden polynomial function $poly(m, n)$ is not specified in [13] and might be rather large.

5 The $O(4.42^k + poly(m, n))$ algorithm

In this section, we introduce a small modification of our algorithm described in Sec. 4. By an involved case analysis, we show in the following theorem that our new algorithm has running time $O(4.42^k\,(m + n) + mn)$.

Theorem 3. *The* Minimum-flip Problem *can be solved in time* $O(4.42^k\,(m+n)+mn)$.

Proof. We now give an overview of the case differentiation of our new algorithm and show that it generates a search tree of size at most 4.42^k.

If there are c-vertices of degree at least four in G, our branching strategy for big M-graphs introduced in Sec. 4.1 has a branching number of at most 4.42 as shown in Lemma 7.

If every c-vertex in G is of degree at most three, we distinguish the following cases:

Case 1: Every c-vertex in G has degree two. The Minimum-flip Problem in G can be solved in polynomial time or by our branching strategy introduced in Sec. 4.2 with branching number at most 1.62.

Case 2: There is a c-vertex of degree two occurring in an M-graph. We show in Sec. 5.2 that our branching strategy for big M-graphs has a branching number of at most 4.24 for this case.

Case 3: Every c-vertex occurring in M-graph has degree three. We observe the following subcases:

– *There are two c-vertices with one common neighbor.* Our branching strategy for big M-graphs has a branching number of at most 4, see Sec. 5.2.
– *Every two c-vertices have two common neighbors.* The character graph G has to be a graph similar to the graph shown in Fig. 6 and the Minimum-flip Problem for G can be solved in polynomial time, as shown in Lemma 8, or G contains four t-vertices and at most four c-vertices as shown in Figure 7 and the problem can even be solved in constant time.

Thus, our algorithm generates a search tree of size at most $O(4.42^k)$. As mentioned in Sec. 4, the execution of data reduction takes $O(mn)$ time. By saving the degree of every vertex in G, our new algorithm also needs $O(m + n)$ time for execution the data reduction and the cases differentiation described above in every recursion call. Thus, the running time of our algorithm is $O(4.42^k\,(m + n) + mn)$. □

We again can get rid of the polynomial factor $(m + n)$ in the running time by using the reduction to a $O(k^3)$ kernel [13] during the course of the search tree algorithm. This results in the running time $O(4.42^k + poly(m, n))$ as claimed.

In the remaining part of this section, we will prove the running time of our algorithm in detail.

5.1 Solving instances with c-vertices of degree at least four

We now assume that G contains a c-vertex of degree at least four. The following lemma holds.

Lemma 7. *If there is a c-vertex of degree at least four in G, the branching strategy for big M-graphs described in Sec. 4.1 has branching number* 4.42 *when branching into subcases eliminating a big M-graph containing the c-vertex with the maximum degree.*

Proof. Assume that G contains a c-vertex of degree at least four. Let c be a c-vertex with maximum degree in G. It holds that c has degree at least four, and there must exist a c-vertex c' in the c-neighborhood of c with $N(c') \nsubseteq N(c)$, otherwise c is removed from G by data reduction Rule 1. In the following, we make a case analysis by means of the degree of c', and show that our branching strategy for big M-graphs described in Sec. 4.1 has branching number at most 4.42 when branching into subcases eliminating the big M-graph containing c and c'. Assume that c has degree four, so c' can have degree two, three, or four.

Case 1: c' has degree two. G contains the subgraph shown in Figure 4(a) as an induced subgraph. Since every pair $\{c', t\}$ for $t \notin N(c')$ is set to forbidden by the data reduction, our branching leads to the branching vector $(3, 1, 1, 1, 1)$, which corresponds to the branching number 4.07.

Case 2: c' has degree three. We distinguish two subcases: c and c' have one common neighbor, c and c' have two common neighbors, see Figure 4(c). Branching on the big M-graph induced by c and c', our branching strategy results in branching vector $(3, 3, 3, 3, 3, 3, 3, 3, 1, 1, 2, 2, 2, 2)$ with branching number 3.68, see Figure 4(c) left; or branching vector $(2, 2, 2, 2, 2, 2, 2, 2, 1, 1)$ with branching numbers 4, see Figure 4(c) right.

Case 3: c' has degree four. For the cases that c and c' have one or two common neighbors, the branching number of our branching strategy is dominated by the corresponding cases where c' has degree three. If c and c' have three common neighbors, G contains the subgraph showed in Figure 4(b) as induced subgraph and our branching strategy has a branching number of 4.42.

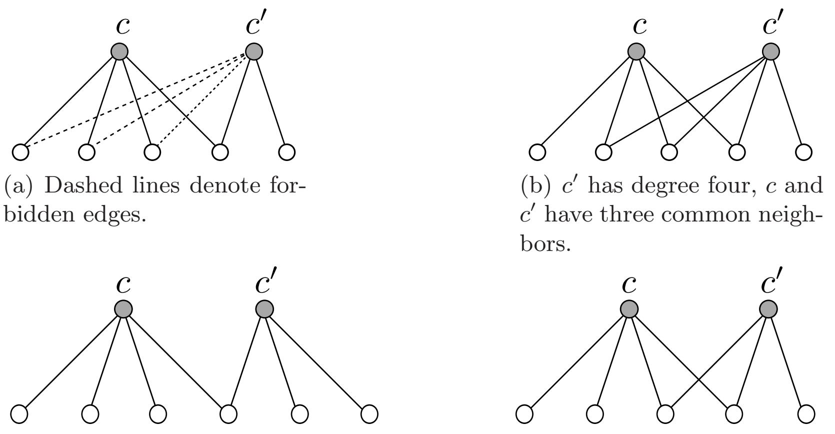
If c has degree five or more, c' can have degree larger than four, but the big M-graph induced by c, c' and their neighborhood contains one of the big M-graphs analyzed above as induced subgraph. This leads to the branching number of 4.42. □

Since degrees of all vertices in G is established, the c-vertex with maximum degree and the big M-graph containing this c-vertex can be found in $O(m + n)$ time.

5.2 Solving instances with c-vertices of degree at most three

In the following, we describe how to solve problem instances where all c-vertices have degree at most three. If every c-vertex is of degree two, the problem can be solved in polynomial time as mentioned in Sec. 4.2 or by our branching strategy, also introduced there, with branching number 1.62. We now assume that there are c-vertices of degree three in G.

Assume that there exists a c-vertex c of degree two that occurs in an M-graph. Let c' be the other c-vertex in this M-graph. Vertex c' can have degree two or degree three. In any case, we do not have to insert edges incident to c, since c is of degree two. We can infer the



(a) Dashed lines denote forbidden edges.

(b) c' has degree four, c and c' have three common neighbors.

(c) *Left:* c and c' have one common neighbor. *Right:* c and c' have two common neighbors.

Fig. 4. c-vertex of degree at least four in G

following: If c' has degree two, we do not have to insert edges, so the branching strategy for big M-graphs has branching vector $(1, 1, 1, 1)$ with branching number 4, see Figure 5 left. If c' has degree three, this branching strategy has branching vector $(1, 1, 1, 1, 2)$ that leads to a branching number of 4.24, see Figure 5 right.

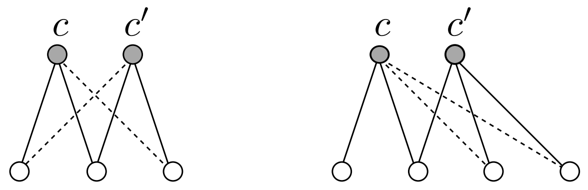


Fig. 5. Every c-vertex has degree at most three and a c-vertex of degree two occurs in an M-graph. Dashed lines will not be inserted by the branching strategy.

In the following, we assume that every c-vertex that occurs in an M-graph has degree three. It must holds that each c-vertex of degree two must have either zero or two common neighbors with other c-vertex. If there are two c-vertices c, c' with one common neighbor, it holds that $|X(c, c')| = 2$, $|Y(c, c')| = 1$ and $|Z(c, c')| = 2$. When branching into subcases eliminating the big M-graph containing c and c', the branching strategy introduced in Sec. 4.1 results in the branching vector $(2, 2, 2, 2, 1, 1, 2, 2, 2, 2)$, which corresponds to a branching number of 4.

We consider the last case where *every pair of c-vertices in G must have two or three common neighbors.* Since each c-vertex has three neighbors, all c-vertices with three common neighbors have the same neighborhood and are thus merged into a single c-vertex by our data reduction. This implies that *every pair of c-vertices have two common neighbors.* This case can again be divided into two subcases:

Case 1: There are two t-vertices that are connected to all c-vertices in G. Let t', t'' be these two t-vertices. An arbitrary c-vertex c_i must have t', t'' and a t-vertex t_i in his neighborhood and the neighborhood of t_i contains only c_i, since c-vertices with the same neighborhood are merged into a single one. See Figure 6. According to the following lemma, the Minimum-flip Problem can be solved in polynomial time in this case.

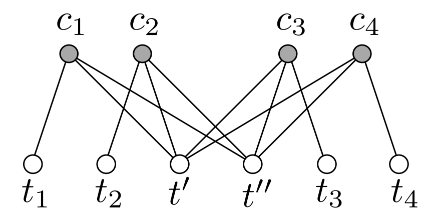


Fig. 6. t-vertices t', t'' are the two common neighbors of all c-vertices.

Lemma 8. *If every c-vertex in G has degree at most three and the neighborhood of every c-vertex c_i with degree three shares two common t-vertices t' and t'', i.e. $N(c_i) = \{t_i, t', t''\}$ (see Fig. 6), then there is an optimal solution that deletes all edges $\{c_i, t_i\}$ except for one edge $\{c_{max}, t_{max}\}$ with $w(c_{max}, t_{max}) = \max_i\{w(c_i, t_i)\}$.*

Proof. Any c-vertex of degree two can only have t' and t'' in his neighborhood and never occurs in an M-graph even if we delete any edge (c_i, t_i). Recall that all edge modifications associated with the same c-vertex must have the identical cost. Let w_i denote the cost of edge modifications associated with c-vertex c_i. Without lost of generality, assume that $w_i \leq w_j$ if $i < j$. We prove this lemma by induction.

If there are only two c-vertices c_{m-1} and c_m in G, deleting $\{c_{m-1}, t_{m-1}\}$ is clearly an optimal solution for the minimum-flip consensus trees in G.

Let G' be the subgraph of G without c_1. Assume that the optimal solution for G' deletes all edges $\{c_i, t_i\}$ for $2 \leq i \leq m-1$. The cost of the optimal solution for G' is $k_{opt}(G') = \sum_{i=2}^{m-1} w_i$.

Assume that there is an optimal solution with cost $k_{opt}(G) < k_{opt}(G') + w_1$ for G. Let G_{opt} be the output graph of this optimal solution for G. In the following we show that there is a solution for G' with cost smaller than $k_{opt}(G')$, which is a contradiction to the assumption that the cost of the optimal solution for G' is $k_{opt}(G')$.

A solution for G' can be derived from G_{opt} by undoing edge modifications associated with c_1, if some is executed, and then removing c_1 from G_{opt}. It is clear that the resulting graph is a solution for G'.

If the optimal solution for G did execute some edge modifications associated with c_1, undoing these edge modifications reduces $k_{opt}(G)$ to at most $k_{opt}(G) - w_1$, which is smaller than $k_{opt}(G')$. This is a contradiction to the assumption that the cost of the optimal solution for G' is $k_{opt}(G')$.

If no edge modification associated with c_1 was executed by the optimal solution for G, at least one modification associated with each c_i for $2 \leq i \leq m$ had to be executed to eliminate all (big) M-graphs containing c_1 and c_i. Therefore, the cost $k_{opt}(G)$ of the optimal solution for G can be lower-bounded by $\sum_{i=2}^{m} w_i$, whereby deleting all edges $\{c_i, t_i\}$ for $1 \leq i \leq m-1$ results in a solution for G with cost $\sum_{i=1}^{m-1} w_i \leq \sum_{i=2}^{m} w_i \leq k_{opt}(G)$. Since the equality holds if and only if $w_i = w_j$ for $1 \leq i, j \leq m$, there is no solution for G with cost smaller than $\sum_{i=2}^{m-1} w_i + w_1 = k_{opt}(G') + w_1$ and since our algorithm computes a solution for G with this cost, it computes the optimal solution for G. □

Case 2: There are no two t-vertices that are common neighbors of all c-vertices.

In the following, we show that in this remaining case, the graph G consists of four t-vertices and at most four c-vertices as shown in Fig. 7.

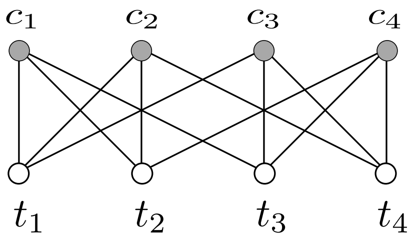


Fig. 7. Every c-vertex is of degree three, and every two c-vertices have two common neighbors, and there are no two t-vertices that are neighbors of all c-vertices.

Since we merge c-vertices with the same neighborhood, there are no c-vertices with three common neighbors in G. If G contains at least two c-vertices, then G must also contain at least four t-vertices, otherwise the two c-vertices are merged by our data reduction. Suppose G contains at least five t-vertices. Let $\{t_1, t_2, t_3, t_4, t_5\}$ be five t-vertices in G. Since every c-vertex in G is of degree three and every two c-vertices in G have two common neighbors, there must be at least three c-vertices in the neighborhoods of these five t-vertices. Let c_1, c_2, c_3 be these three c-vertices, where $N(c_1) = \{t_1, t_2, t_3\}$ and $N(c_2) = \{t_2, t_3, t_4\}$. Obviously, t_5 has to belong to $N(c_3)$. Since every two c-vertices in G must have two common neighbors, the others two neighbors of c_3 must be t_2, t_3. If G does contain further c-vertices, t_2, t_3 must be common neighbors of every c-vertex in G by the same argumentation. This implies that t_2, t_3 are common neighbors of every c-vertex and we are done. Therefore, we may assume that, G contains only four t-vertices. Since every c-vertex in G is of degree three, there are at most $\binom{4}{3}$ c-vertices in G after merging c-vertices with the same neighborhood. See Fig.7 for an illustration of this case. Thus, G contains only four t-vertices and at most four c-vertices and the Minimum-flip Problem in G is solvable in constant time. This finishes the tedious case analysis of the proof that our algorithm solves the Minimum-flip Problem in time $O(4.42^k\,(m+n)+mn)$.

We conjecture that using the technique introduced by Gramm et al. [10] might lead to search tree algorithms with better *theoretical* running times. But these algorithms mostly do not have practical applicability.

6 Algorithm engineering

In the course of algorithm design, we found some improvements that do not affect the theoretical worst-case running time or even increase the polynomial factor but as they manage to prune the search tree, they are highly advisable in practice. These are a few heuristic improvements we included in our implementation.

Treat connected components separately If a given character graph is not connected or decomposes during the execution of the algorithm, we compute the solutions for each of its connected components separately because connecting different connected components never deletes an M-graph.

Avoid futile program calls If there are two c-vertices c_i, c_j such that

$$\min\{X(c_i,c_j), Y(c_i,c_j), Z(c_i,c_j)\} > k$$

holds, we know that it is impossible to solve the current instance (see Corollary 2). Therefore, whenever we find such an M-graph we abort the current search tree branch and call the algorithm with an appropriately increased parameter, thus skipping program runs that are doomed to failure.

Avoid redundant search tree branches When executing an edit operation in a big M-graph, we fix the outcome of the operation, that is, whenever we insert an edge, this edge is set to "permanent" and when we delete an edge, it is set to "forbidden". With this technique we make sure that edit operations are not undone later in the search tree.

Try promising search tree branches first In the first part of our branching strategy, branching on a big M-graph F_{M} with c-vertices c_i, c_j leads to $2^{|X(c_i,c_j)|} + 2^{|Y(c_i,c_j)|} + 2^{|Z(c_i,c_j)|}$ branches. It is likely that a minimum solution destroys the F_{M} with as few edge modifications as possible. As we use depth-first search and stop when we find a solution, we branch on the edges incident to the smallest of sets X, Y, Z first.

Calculate branching numbers in advance When dealing with big M-graphs, we save, for each pair of c-vertices, the branching number corresponding to a branching at the F_{M} associated with these vertices in a matrix. The minima of each row are saved in an extra column to allow faster searching for the overall minimum. We use a similar technique to deal with the weighted graph in the second part of the algorithm.

Note that due to the expense of development and a hidden large polymonimal factor of the kernelization in [13], we do not implement this kernelization. Furthermore, the polynomial factor in the running time proved in Theorem 2 and Theorem 3 cannot hold, when applying the abovementioned heuristic improvements, since initializing the matrix used to calculate the branching numbers takes time $O(m^2 n)$ and updating this matrix needs time $O(mn)$ in each recursive call. While initializing or updating this matrix, we also check if the data reduction rules can be applied. Testing for futile program calls and redundant search tree branches can be executed in the same time. Altogether, the running time of our algorithms with the abovementioned heuristic improvements are $O(4.83^k\, mn + m^2 n)$ and $O(4.42^k\, mn + m^2 n)$, respectively. Despite that, the heuristic improvements lead to drastically reduced running times in practice.

7 Experiments

The only difference between our 4.83^k and 4.42^k algorithms is that when dealing with the special case of G shown in Figures 6 or 7, the 4.42^k algorithm uses the corresponding polynomial time procedures instead of the branching strategy for big M-graphs. Since these special cases do not occur often, we believe that in practice, our 4.42^k algorithm is not faster than our 4.83^k algorithm. Therefore, we just implemented our 4.83^k algorithm, and compared it against Chen et al.'s 6^k algorithm [6]. Both algorithms were implemented in Java. Computations were done on an AMD Opteron-275 2.2 GHz with 6 GB of memory running Solaris 10.

Each program receives a binary matrix as input and returns a minimum set of flips needed to solve the instance. We start the program repeatedly and increase k by one until a

Dataset	$\lvert V_t \rvert$	$\lvert V_c \rvert$	# flips	avg. k	time 4.83^k	time 6^k
Marsupials	21	20	10	9.6	9.5 s	2 h
			12	10.7	25.5 s	> 10 h*
			14	13.2	3 min	> 10 h
			16	15.4	12 min	> 10 h
			18	17	47 min	> 10 h
			20	18.9	3.3 h	> 10 h
Marsupials	51	50	10	10	17 s	19 h
			12	10.5	30 s	> 10 h
			14	12.5	2.3 min	> 10 h
			16	15.5	50 min	> 10 h
			18	17.5	3 h	> 10 h
			20	19	8 h	> 10 h
Tex (Bacteria)	97	96	10	9.7	17 s	59.3 h
			12	11.9	12.5 min	> 10 h
			14	13.9	18 min	> 10 h
			16	15.4	1.1 h	> 10 h
			18	17.3	4 h	> 10 h
			20	19.7	10.3 h*	> 10 h

Table 1. Comparison of running times of our 4.83^k algorithm and the 6^k algorithm. $|V_t|$ and $|V_c|$ denote the number of t- and c-vertices, respectively. # flips is the number of perturbations in the matrix, whereas k is the true number of flips needed to solve the instance. Each row corresponds to ten datasets. *Six out of ten computations were finished in less than ten hours.

solution is found. As soon as we find a solution with at most k flips, the program is aborted instantly and the solution is returned without searching further branches. All data reduction rules and heuristic improvements described in Sec. 3 and Sec. 6 were implemented and, where applicable, also used in the 6^k algorithm. For our experiments we used matrix representations of real phylogenetic trees, namely two phylogenetic trees of marsupials with 21 and 51 taxa (data provided by Olaf Bininda-Emonds) and one tree of 97 bacteria computed using Tex protein sequences (data provided by Lydia Gramzow). Naturally, these matrices admit perfect phylogenies.

We perturbed each matrix by randomly flipping different numbers of entries, thus creating instances where the number of flips needed for resolving all M-graphs in the corresponding character graph is at most the number of perturbations. For each matrix representation and each number of perturbations, we created ten different instances and compared the running times of both algorithms on all instances. In many datasets we created, it was possible to solve the instance with a smaller number of flips.

Each instance was allowed ten hours of computation. For the largest instances with only ten flips, we ignored this running time constraints to show the performance of the 6^k algorithm. The results of the computations are shown in Table 1. When the average running time was below ten hours, all instances were finished in less than ten hours. When the average was more than ten hours, all ten instances took more than ten hours, except for the small Marsupial datasets with $k = 12$ for the 6^k algorithm and the Tex datasets with $k = 20$ for our algorithm. In both cases, six of ten instances were solved.

Our experiments show that our method is significantly faster than Chen et al.'s algorithm for all instances. We observed that, on average, increasing k by one resulted in about 2.2-fold running time for a program call of our algorithm and 5-fold running time for the 6^k algorithm. We believe that the reason for the factor of 2.2 is that big M-graphs can be fairly large in practice, so that the real branching number is close to two as analyzed in Sec. 4.1.

8 Conclusion

We have presented two new refined fixed-parameter algorithms for the Minimum-Flip Problem. These methods improve the worst-case running time for the exact solution of this problem mainly by downsizing the search tree from $O(6^k)$ to $O(4.83^k)$ and $O(4.42^k)$. The experiments show that in practice the difference in running times is by far larger than one would expect from the worst-case analysis. Our algorithms outperformed the 6^k algorithm dramatically. We believe that this is a big step towards computing exact solutions efficiently. Even if our program may never be fast enough to solve very large instances, it is certainly useful for tuning and evaluating heuristic algorithms such as Chen *et al.*'s heuristic and approximation algorithms [5, 6].

To create not only consensus trees but also arbitrary supertrees, we have to consider a version of the Minimum-Flip Problem where, besides zeros and ones, many matrix entries are '?' (unknown) and we have to create a matrix without question marks that admits a perfect phylogeny with as few flips of zeros and ones as possible. This problem is called the Minimum-Flip Supertree Problem problem [6]. It is an interesting open question if this problem is fixed-parameter tractable with respect to the minimum number of flips. An algorithm that can handle this problem would make an interesting tool in computational biology. As the next step of our research on the Minimum-Flip Problem, we will investigate solving the Minimum-Flip Supertree problem exactly and compare its performance to the MRP approach [2]. Note that our algorithm can be slightly modified to compute exact solution for the Minimum-Flip Supertree Problem, but we cannot give any guarantee about the running time. Furthermore, we do not make the implementation of our algorithm publicity available at the moment, but it can be obtained from us by request.

Acknowledgment

All programming and computational experiments were done by Patrick Seeber. We thank Olaf Bininda-Emonds and Lydia Gramzow for providing the datasets. Q.B.A Bui gratefully acknowledges financial support from the DFG, research group PABI (Parameterized Algorithms in Bioinformatics) BO 1910/5.

Bibliography

[1] E. N. Adams III. Consensus techniques and the comparison of taxonomic trees. *Syst. Zool.*, 21(4):390–397, 1972. doi: 10.2307/2412432.

[2] O. R. P. Bininda-Emonds, editor. *Phylogenetic Supertrees: Combining Information to Reveal the Tree of Life*, volume 4 of *Computational Biology Series*. Kluwer Academic, 2004.

[3] D. Bryant and M. A. Steel. Extension operations on sets of leaf-labelled trees. *Adv. Appl. Math.*, 16(4):425–453, 1995. doi: 10.1006/aama.1995.1020.

[4] L. Cai. Fixed-parameter tractability of graph modification problems for hereditary properties. *Inform. Process. Lett.*, 58(4):171–176, 1996. ISSN 0020-0190. doi: 10.1016/0020-0190(96)00050-6.

[5] D. Chen, O. Eulenstein, D. Fernández-Baca, and J. G. Burleigh. Improved heuristics for minimum-flip supertree construction. *Evol. Bioinform. Online*, 2:347–356, 2006. URL http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2674677/pdf/ebo-02-391.pdf.

[6] D. Chen, O. Eulenstein, D. Fernández-Baca, and M. Sanderson. Minimum-flip supertrees: Complexity and algorithms. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 3(2):165–173, 2006. doi: 10.1109/TCBB.2006.26. URL http://dx.doi.org/10.1109/TCBB.2006.26.

[7] W. Day, D. Johnson, and D. Sankoff. The computational complexity of inferring rooted phylogenies by parsimony. *Math. Biosci.*, 81(1):33–42, 1986. doi: 10.1016/0025-5564(86)90161-6.

[8] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, Berlin, 1999.

[9] H. N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proc. of ACM-SIAM Symposium on Discrete Algorithms (SODA 1990)*, pages 434–443. Society for Industrial and Applied Mathematics, 1990.

[10] J. Gramm, J. Guo, F. Hüffner, and R. Niedermeier. Automated generation of search tree algorithms for hard graph modification problems. *Algorithmica*, 39(4):321–347, 2004. doi: 10.1007/s00453-004-1090-5.

[11] D. Gusfield. Efficient algorithms for inferring evolutionary trees. *Networks*, 21(1): 19–28, 1991. doi: 10.1002/net.3230210104.

[12] S. Kannan, T. Warnow, and S. Yooseph. Computing the local consensus of trees. In *Proc. of Symposium on Discrete Algorithms (SODA 1995)*, pages 68–77. Society for Industrial and Applied Mathematics, 1995.

[13] C. Komusiewicz and J. Uhlmann. A cubic-vertex kernel for flip consensus tree. In *Proc. of Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2008)*, volume 2 of *Leibniz International Proceedings in Informatics*, pages 280–291. Dagstuhl, 2008. doi: 10.4230/LIPIcs.FSTTCS.2008.1760.

[14] M. P. Ng and N. C. Wormald. Reconstruction of rooted trees from subtrees. *Discrete Appl. Math.*, 69(1-2):19–31, 1996.

[15] R. Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, 2006.

[16] R. Niedermeier and P. Rossmanith. A general method to speed up fixed-parameter-tractable algorithms. *Inform. Process. Lett.*, 73:125–129, 2000.

[17] C. Semple and M. Steel. A supertree method for rooted trees. *Discrete Appl. Math.*, 105(1-3):147–158, 2000. ISSN 0166-218X. doi: 10.1016/S0166-218X(00)00202-X.