An Improved Fixed-Parameter Algorithm for Minimum-Flip Consensus Trees

Sebastian Böcker, Quang Bao Anh Bui, and Anke Truss

Lehrstuhl für Bioinformatik, Friedrich-Schiller-Universität Jena,
Ernst-Abbe-Platz 2, 07743 Jena, Germany
{boecker,bui,truss}@minet.uni-jena.de

Abstract. In computational phylogenetics, the problem of constructing a consensus tree for a given set of 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. In its graph-theoretical formulation, the problem is as follows: Given a bipartite graph $G = (V_t \cup V_c, E)$, the problem is to find a minimum set of edge modifications such that the resulting graph has no induced path with four edges which starts and ends in V_t.

We present a fixed-parameter algorithm for the Minimum-Flip Problem with running time $O(4.83^k\,(m+n) + mn)$ for n taxa, m characters, and k flips. Additionally, we discuss several heuristic improvements. We also report computational results on phylogenetic data.

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 which 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 [12, 3], 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 [14]. 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. Problem is NP-complete [8], and so is MRP. The matrix representation with flipping (MRF) supertree method also uses a binary matrix representation of the input trees [5]. 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. [6] indicate that MRF is superior to MRP and other common approaches for supertree construction, such as MinCut supertrees [14].

If all input trees share the same set of taxa, the supertree is called a consensus tree [1, 11]. As in the case of 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 between taxa. In this case, we can construct the corresponding *perfect phylogeny* in $O(mn)$ time for n taxa and m characters [10]. 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 Minimum-Flip Problem of finding the minimum set of flips which make a matrix compatible, is NP-hard [6, 7]. Based on a graph-theoretical interpretation of the problem and the forbidden subgraph paradigm of Cai [4], Chen et al. introduce a simple fixed-parameter algorithm with running time $O(6^k\, mn)$, where k is the minimum number of flips [6, 7]. Furthermore, the problem can be approximated with approximation ratio $2d$ where d is the maximum number of ones in a column [6, 7].

Our contributions. We introduce a refined fixed-parameter algorithm for the Minimum-Flip Problem with $O(4.83^k\,(m+n)+mn)$ running time, and discuss some heuristic improvements to reduce the practical running time of our algorithm. To evaluate the performance and to compare it to the fixed-parameter algorithm of Chen et al., we have implemented both algorithms and evaluate them on perturbed matrix representations of phylogenetic trees. Our 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

Throughout this paper, let n be the number of taxa characterized by m 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 [10] introduces an algorithm which checks if a matrix M admits a perfect phylogeny and, if possible, constructs the corresponding phylogenetic tree in total running time $O(mn)$.

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

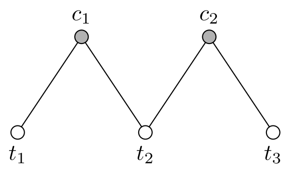

Fig. 1. An M-graph. Grey vertices denote characters, white vertices denote taxa.

M into a matrix which admits a perfect phylogeny. The corresponding decision problem is to check whether there exists a solution with at most k flips. 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 a $n \times m$ 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 provides an essential characterization with regard to the graph-theoretical modeling of the Minimum-Flip Problem.

Theorem 1 (Chen et al. [7]). *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, which transform the character graph of the input matrix into an M-free bipartite graph. Using this characterization, Chen et al. [6, 7] 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 $Z(c_i, c_j) := N(c_j) \setminus N(c_i)$. We call c_i and c_j *c-neighbors* if and only if $Y(c_i, c_j)$ is not empty.

3 The Algorithm

We present a search tree algorithm largely based on observations on the structure of intersecting M-graphs. First, we use a set of reduction rules to cut down the size of G, see Sect. 3.1 below. As long as there are c-vertices of degree three or higher in G, we use the branching strategy described in Sect. 3.2. If there are no such vertices left, we can use a simplified branching strategy described in Sect. 3.3. In the beginning of every recursion call of our search tree algorithm we execute the data reduction described in the following section.

3.1 Data Reduction

When the algorithm receives a character graph G as input, it is reduced with respect to the following simple reduction rules:

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.

We verify the correctness of these reduction rules, starting with Rule 1. 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 which deletes edges incident to c in G. If we execute all edit operations of the optimal solution except deletions of edges incident to c, we also obtain an M-free graph, since every M-graph which does not contain c is destroyed by edit operations 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 $O(m + n)$.

3.2 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 1. *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 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

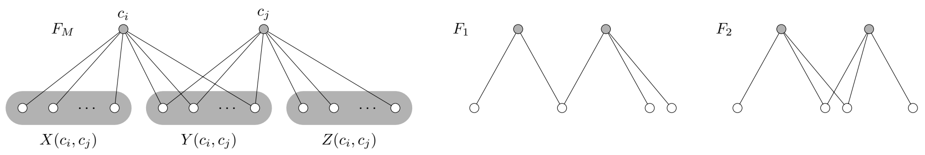

Fig. 2. The big M-graph F_{M} and its special cases F_1 and F_2

removed from G by the data reduction. Let t_k be a common neighbor of c_i and c_j which has to exist since c_i and c_j are c-neighbors. There also exists a neighbor t_i of c_i which is not a neighbor of c_j. If t_i is the only one t-vertex, which is neighbor of t_i but not of t_j, then t_i and t_j must share an another common neighbor t'_k besides t_k (since t_i has degree of 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 neighbor of t_i but not of t_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 1, 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 2. *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 2 is obvious.

Lemma 2 leads us 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 2, 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 edit-operations. 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 edit operations 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 2, at least $\min\{x, y, z\}$ edit-operations must be executed to eliminate all M-graphs in F_{M}. Our branching strategy has branching vector

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

which leads to a branching number of 4.83 as shown in the following lemma (see [13] for details on branching vectors and branching numbers).

Lemma 3. *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

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

Considering $\frac{b}{2}$ 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. $\square$

From Lemma 2 we infer an interesting property. When we take into consideration that we need an edit operation for each vertex we remove from set X, Y, or Z, the following corollary is a straightforward observation.

Corollary 1. *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 Sect. 4.

Corollary 1 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 edit operations which lower k to a greater extent. Therefore the branching number of our strategy goes to 2 for large x, y, z, and this is confirmed by the growth of running times in our computational experiments (see Sect. 5).

3.3 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 1, if a character graph G reduced with respect to our data reduction does not contain any big M-graph, every c-vertex in G has degree two. In this case we use the branching strategy based on the following lemma to transform G into an M-free character graph.

Lemma 4. *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.*

Proof. Let G be a character graph where all c-vertices have degree two and G' be the resulting graph of an optimal solution for G where we did add new edges to G and $\{c, t\}$ be such a new edge.

In the following we show that there is another optimal solution for G without any edge insertion. Let t_i, t_j be the t-vertices connected with c. 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 removing one of the edges $\{c, t_i\}$ or $\{c, t_j\}$, vertex c has degree one and cannot be vertex of any M-graph. Therefore the resulting graph is still M-free and the number of edit operations does not increase since we swap an insertion for only one deletion. Hence, edge insertions are not necessary for optimal solutions when all c-vertices of the character graph have degree two. $\square$

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 number of common neighbors of t_1, t_2 in G, see Fig. 3.

On the weighted graph G_w and the number k of remaining edit operations, 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.

Since it is unknown if the abovementioned problem can be solved in polynomial time, we used the fixed-parameter algorithm described in the following text to deal with this problem. 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

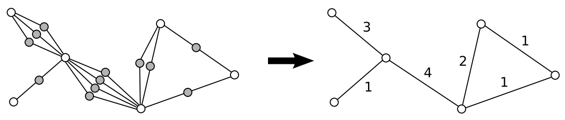

Fig. 3. Left: When all c-vertices (gray) have degree two, we can regard each c-vertex with its incident edges as a single edge in a multigraph whose vertex set is the set of t-vertices (white). Right: We merge all those "edges" between two t-vertices into a single weighted edge whose weight equals the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

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 which 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.

Let us 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 which are incident to a vertex in e.

Now we pick an edge e which 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 which 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.

We prove in the following theorem 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 Sect. 3.2 with branching number 4.83, in the second case it executes the branching strategy in Sect. 3.3 with branching number 1.62. Therefore, the size of the search tree is $O(4.83^k)$. All in all, the running time of the algorithm is $O(4.83^k\,(m + n) + mn)$. □

4 Algorithm Engineering

In the course of algorithm design, we found some improvements which 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 the course of implementation.

Treat connected components separately. If a given character graph is not connected or decomposes in the course 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 $\min\{X(c_i, c_j), Y(c_i, c_j), Z(c_i, c_j)\} > k$ holds for two c-vertices c_i, c_j, we know that it is impossible to solve the current instance (see Corollary 1). 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 which 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 in order 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.

The polynomial factor in the running time proved in Theorem 2 cannot be hold when applying the abovementioned heuristic improvements, since initializing the matrix used to calculate the branching numbers takes time $O(m^2n)$ and updating this matrix needs time $O(mn)$ in each recursion 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 algorithm with the abovementioned heuristic improvements is $O(4.83^k\, mn + m^2n)$. Despite that, the heuristic improvements lead to drastically reduced running times in practice.

Table 1. Comparison of running times of our $O(4.83^k)$ algorithm and the $O(6^k)$ algorithm. $|V_t|$ and $|V_c|$ denote the number of t- and c-vertices, respectively. # flips is the number of perturbances 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 under ten hours.

Dataset	$\|V_t\|$	$\|V_c\|$	# 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

5 Experiments

To evaluate in how far our improved branching strategy affects running times in practice, we compared our algorithm against Chen et al.'s $O(6^k\, mn)$ algorithm [6, 7]. 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. The parameter need not be given as the program starts with calling the algorithm with $k = 0$ and repeatedly increases k by one until a solution is found. As soon as it finds a solution with at most k flips, the program call is aborted instantly and the solution is returned without searching further branches. All data reduction rules and heuristic improvements described in Sect. 3.1 and 4 were used for our algorithm and, if applicable, also for the $O(6^k\, mn)$ algorithm. For our experiments we used matrix representations [9] 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 perturbances. For each matrix representation and each number of perturbances 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 dataset was allowed ten hours of computation. Running times for the $O(6^k\,mn)$ algorithm for ten flips on the two larger datasets were calculated despite this restriction to show the order of magnitude. The results of the computations are summed up in Table 1. When the average running time was below ten hours, all instances were finished in less then 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 $O(6^k\,mn)$ 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 constantly significantly faster than Chen et al.'s algorithm. In the course of computations 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 $O(6^k)$ search tree algorithm. The reason for the factor of 2.2 is probably that big M-graphs can be fairly large in practice such that the real branching number is close to two as analyzed in Sect. 3.2.

6 Conclusion

We have presented a new refined fixed-parameter algorithm for the Minimum-Flip Problem. This method improves the worst-case running time for the exact solution of this problem mainly by downsizing the search tree from $O(6^k)$ to $O(4.83^k)$. 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 algorithm outperformed the $O(6^k)$ algorithm dramatically. We believe that this a big step towards computing exact solutions efficiently.

Since the Minimum-Flip Problem is fixed-parameter tractable with respect to the minimum number of flips, a problem kernel must exist [13]. Finding a kernelization procedure is a natural next step in the theoretical analysis, and, to our expectation, may also greatly improve running times. 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 [6, 7].

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, a considerable amount of matrix entries are '?' (unknown) and we have to create a matrix without question marks which admits a perfect phylogeny with as few flips of zeros and ones as possible. It is an interesting open question if this problem is fixed-parameter tractable with respect to the minimum number of flips. An algorithm which can handle this problem would make an interesting tool in computational biology.

Acknowledgment

All programming and experiments were done by Patrick Seeber. We thank Olaf Bininda-Emonds and Lydia Gramzow for providing the datasets.

References

1. Adams III, E.N.: Consensus techniques and the comparison of taxonomic trees. Syst. Zool. 21(4), 390–397 (1972)
2. Bininda-Emonds, O.R.: Phylogenetic Supertrees: Combining Information to Reveal the Tree of Life. Computational Biology Book Series, vol. 4. Kluwer Academic, Dordrecht (2004)
3. Bryant, D., Steel, M.A.: Extension operations on sets of leaf-labelled trees. Adv. Appl. Math. 16(4), 425–453 (1995)
4. Cai, L.: Fixed-parameter tractability of graph modification problems for hereditary properties. Inf. Process. Lett. 58(4), 171–176 (1996)
5. Chen, D., Diao, L., Eulenstein, O., Fernández-Baca, D., Sanderson, M.: Flipping: A supertree construction method. In: Bioconsensus. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 61, pp. 135–160. American Mathematical Society, Providence, RI (2003)
6. Chen, D., Eulenstein, O., Fernández-Baca, D., Sanderson, M.: Supertrees by flipping. In: H. Ibarra, O., Zhang, L. (eds.) COCOON 2002. LNCS, vol. 2387, pp. 391–400. Springer, Heidelberg (2002)
7. Chen, D., Eulenstein, O., Fernandez-Baca, D., Sanderson, M.: Minimum-flip supertrees: Complexity and algorithms. IEEE/ACM Trans. Comput. Biol. Bioinform. 3(2), 165–173 (2006)
8. Day, W., Johnson, D., Sankoff, D.: The computational complexity of inferring rooted phylogenies by parsimony. Math. Biosci. 81, 33–42 (1986)
9. Farris, J., Kluge, A., Eckhardt, M.: A numerical approach to phylogenetic systemetics. Syst. Zool. 19, 172–189 (1970)
10. Gusfield, D.: Efficient algorithms for inferring evolutionary trees. Networks 21, 19–28 (1991)
11. Kannan, S., Warnow, T., Yooseph, S.: Computing the local consensus of trees. In: Proc. of Symposium on Discrete Algorithms (SODA 1995) (1995)
12. Ng, M.P., Wormald, N.C.: Reconstruction of rooted trees from subtrees. Discrete Appl. Math. 69(1–2), 19–31 (1996)
13. Niedermeier, R.: Invitation to Fixed-Parameter Algorithms. Oxford University Press, Oxford (2006)
14. Semple, C., Steel, M.: A supertree method for rooted trees. Discrete Appl. Math. 105(1–3), 147–158 (2000)