Drawing Binary Tanglegrams: An Experimental Evaluation

Martin Nöllenburg* Markus Völker* Alexander Wolff† Danny Holten†

Abstract

A *tanglegram* is a pair of trees whose leaf sets are in one-to-one correspondence; matching leaves are connected by inter-tree edges. In applications such as phylogenetics or hierarchical clustering, it is required that the individual trees are drawn crossing-free. A natural optimization problem, denoted *tanglegram layout problem*, is thus to minimize the number of crossings between inter-tree edges.

The tanglegram layout problem is NP-hard even for complete binary trees, for general binary trees the problem is hard to approximate if the Unique Games Conjecture holds. In this paper we present an extensive experimental comparison of a new and several known heuristics for the general binary case. We measure the performance of the heuristics with a simple integer linear program and a new exact branch-and-bound algorithm. The new heuristic returns the first solution that the branch-and-bound algorithm computes (in quadratic time). Surprisingly, in most cases this simple heuristic is at least as good as the best of the other heuristics.

1 Introduction

In this paper we are interested in evaluating the performance of several recently suggested algorithms for drawing so-called *tanglegrams* [15], that is, pairs of trees whose leaf sets are in one-to-one correspondence. The need to visually compare pairs of trees arises in applications such as phylogenetics or clustering.

In biology, a phylogenetic tree is a (rooted) binary tree that describes a hypothesis of the evolutionary history of a set of species (or *taxa*) that are placed at the leaves of the tree. Each inner node represents a potential ancestor of the taxa at its child nodes. Figure 1 shows an example. There are various tree reconstruction methods available in computational biology that allow biologists to build a set of candidate trees from the DNA or protein sequences of a set of taxa. These trees can be compared using tree-distance measures, but it is also important to graphically depict the trees. A good drawing of the trees helps biologists to recognize and compare different substructures of the trees. This is more than a single number such as a tree distance can achieve. Moreover, tree drawings are often used to communicate the findings of a study to its readers. For this purpose, tanglegrams have established themselves as a visualization method for pairs of related trees [15].

In clustering, our second application, clusters are often computed incrementally: in the beginning each object forms its own cluster, and then, step by step, the pair of clusters that is closest according to some distance measure is joined. Such a hierarchical clustering is naturally represented by a binary tree called *dendrogram*, where elements are represented by the leaves and each inner node of the tree represents a cluster containing the leaves in its subtree. Pairs of dendrograms of the same data stemming from different clustering algorithms or parameter settings can be compared visually using tanglegrams.

From the application point of view it makes sense to insist that (a) the trees under consideration are drawn plane, that is, without edge crossings, (b) each leaf of one tree is connected by an *inter-tree* edge to the corresponding leaf in the other tree, and (c) the number of crossings among the inter-tree edges is minimized. This results in legible tanglegram drawings: each tree is drawn in the usual style and distracting crossings of inter-tree edges are kept to a minimum. Following the bioinformatics literature [13, 15], we call this the *tanglegram layout* problem; Fernau et al. [5] refer to it as *two-tree crossing minimization.* Formally, the problem can be stated as follows.

Tanglegram Layout (TL): Given a pair $\langle S, T\rangle$ of two rooted trees S and T on n leaves and a bijection between their leaf sets, find a tanglegram layout, that is, two plane drawings of S and T, such that

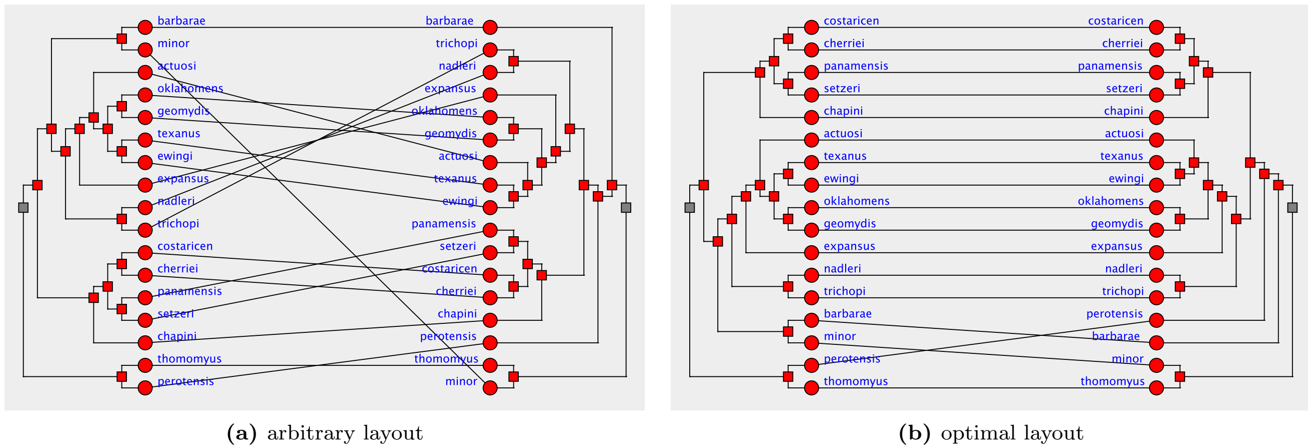
1. the drawing of S is to the left of the line $x = 0$ with all leaves on $x = 0$;
2. the drawing of T is to the right of the line $x = 1$ with all leaves on $x = 1$;
3. the inter-tree edges are drawn as straight-line segments;
4. the number of inter-tree edge crossings is minimum.

Given a tree T, we say that a linear order of its leaves is *compatible* with T if for each node v of T the

*Fakultät für Informatik, Universität Karlsruhe and Karlsruhe Institute of Technology (KIT), Germany. Supported by DFG grant WO 758/4-3. Email: {noellenburg, mvoelker}@iti.uka.de

†Faculteit Wiskunde en Informatica, TU Eindhoven, The Netherlands. Email: {a.wolff, d.h.r.holten}@tue.nl

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.



(a) arbitrary layout

(b) optimal layout

Figure 1: A binary tanglegram of phylogenetic trees for lice of pocket gophers [8].

nodes in the subtree of v form an interval in the order. Note that TL is a purely combinatorial problem: in short, given two trees S and T, TL consists of finding an order σ of the leaves of S compatible with S and an order τ of the leaves of T compatible with T such that the number of inversions between τ and σ is minimum [2, 5]. Let the *crossing number* of a tanglegram $\langle S, T\rangle$ be the minimum number of inter-tree edge crossings of any tanglegram drawing of $\langle S, T\rangle$.

We restrict ourselves to *binary* TL, that is, the case where both given trees are binary. Binary TL is NP-hard [5], even if both trees are complete [2]. Without the restriction to complete trees, the optimization version of binary TL is hard to approximate if the Unique Games Conjecture holds [2].

The contributions of this paper are two-fold. We first give two new exact methods for solving binary TL: a simple integer linear program (ILP) and a branch-and-bound algorithm, see Sections 3.4 and 3.5, respectively. The branch-and-bound algorithm has a variable for the order of the children in each node of the tanglegram. The algorithm exploits the fact that the variables can be chosen independently; it chooses a variable ordering that often yields a very good first solution, whose value can then be used as upper bound to prune many branches of the search tree. The algorithm takes exponential time in general, but yields a fast and simple, yet effective heuristic. The heuristic outputs—in quadratic time—the first solution that the branch-and-bound algorithm finds.

Our second and main contribution is an extensive experimental comparison of this new heuristic and several known heuristics for binary TL (Section 4). We measure the performance of the heuristics with respect to the optimum, which we compute with the above exact methods. Before presenting our comparison, we review related work (Section 2) and describe the algorithms that we compare (Section 3).

2 Related Work

In graph drawing the so-called *two-sided crossing minimization problem* (2SCM) is an important NP-hard problem that occurs when computing layered graph layouts. Such layouts have been introduced by Sugiyama et al. [17] and are widely used for drawing hierarchical graphs. In 2SCM, vertices of a bipartite graph are to be placed on two parallel lines (called *layers*) such that for each vertex on one line all its adjacent vertices lie on the other line. As in TL the objective is to minimize the number of edge crossings provided that edges are drawn as straight-line segments. In one-sided crossing minimization (1SCM) the order of the vertices on one of the layers is fixed. Even 1SCM is NP-hard [4], even if the given graph is a forest of 4-stars [14].

Jünger and Mutzel [10] performed an experimental comparison of exact and heuristic algorithms for both 1SCM and 2SCM. Their main findings were that for 1SCM the exact solution can be computed quickly for up to 60 vertices in the free layer, and for 2SCM an iterated barycenter heuristic is the method of choice for instances with more than 15 vertices in each layer.

The main difference between TL and 2SCM is that in TL, the possible orders of the leaves are limited to those that are compatible with the two input trees. Furthermore, the inter-tree edges are restricted to be a matching of the leaves.

Dwyer and Schreiber [3] studied drawing a series of tanglegrams in 2.5 dimensions, that is, the trees are drawn on a set of stacked two-dimensional planes. They considered a one-sided version of binary TL by fixing the layout of the first tree in the stack, and then, layer-

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

by-layer, computing an optimal leaf order of the next tree with respect to the previous one in $O(n^2 \log n)$ time each. Such a one-sided TL problem is denoted as *one-tree crossing minimization* (1TCM). We include an iterated version of the 1TCM algorithm that alternately fixes one of the trees and optimizes the other as a heuristic in our experimental comparison. Note that the efficient algorithm of Dwyer and Schreiber for 1TCM contrasts the NP-hardness of 1SCM.

Fernau et al. [5] showed how to solve the one-sided version in $O(n \log^2 n)$ time, proved that binary TL is NP-hard, and gave a fixed-parameter algorithm that runs in $O^\star(c^k)$ time, where the $O^\star$-notation ignores polynomial factors, c is a constant that Fernau et al. estimate to be 1024, and k is the crossing number of the given tanglegram. They also made the simple observation that the edges of the tanglegram can be directed from one root to the other. Thus the existence of a planar drawing can be verified using a linear-time upward-planarity test for single-source directed acyclic graphs [1]. Later, apparently not knowing these results, Lozano et al. [13] gave a quadratic-time algorithm for the same special case, to which they refer as *planar tanglegram layout*.

Zainon and Calder [18] described an interactive tree-comparison tool that allows manual and automatic rearrangement of a tanglegram. They aimed at highlighting differences and similarities in the two trees; they did not explicitly minimize the number of inter-tree edge crossings. They implemented two heuristics. The first heuristic starts at the roots of both trees and flips the subtrees of one tree if this increases the number of edges between the aligned subtrees; then it recurses on both pairs of aligned subtrees. The second heuristic minimizes the *triplet difference* between two n-leaf trees over all 2^{2n-2} possible arrangements. The triplet difference counts the number of all three-leaf subsets for which the respective induced subtrees differ in the two trees. They recommended the following semi-automatic approach: use the first heuristic to find a layout of the full trees and then untangle small groups of edges individually using the second (exponential-time) heuristic, followed by some manual fine tuning.

Holten and van Wijk [9] presented a tanglegram visualization tool for the comparison of pairs of large (not necessarily binary) trees. Their tool repeatedly applies the barycenter method [17] to reduce inter-tree crossings (see Section 3.2) and a subsequent edge-bundling technique to reduce visual clutter. The crossing reduction heuristic of Holten and van Wijk is included in our experimental evaluation.

Recently, Buchin et al. [2] showed that binary TL remains NP-hard even if both trees are complete binary trees. For this case they gave an $O(n^3)$-time factor-2 approximation algorithm that recursively splits an instance into subinstances and a simple $O^\star(4^k)$-time fixed-parameter algorithm, where k is as above. They also showed that binary TL is hard to approximate if the Unique Games Conjecture holds. We include in our evaluation a version of their approximation algorithm, that contains improvements for dealing with unbalanced trees.

3 Algorithms

In this section we review the recursive splitting algorithm of Buchin et al. [2] and describe our improvement of their method for unbalanced binary trees. We further describe the hierarchy sorting algorithm of Holten and van Wijk [9] and a variant, and the iterated one-tree crossing minimization algorithm of Dwyer and Schreiber [3]. Finally, a simple integer linear program (ILP) and an exact branch-and-bound algorithm are presented as two new methods that provide optimal solutions for the experiments in Section 4. The branch-and-bound algorithm additionally gives rise to a fast, yet effective heuristic.

3.1 Recursive Splitting Algorithm The main idea behind the recursive splitting algorithm of Buchin et al. [2] is to consider for an instance $\langle S, T\rangle$ the four possible orders of the two subtrees S_1, S_2 of S and T_1, T_2 of T below the roots v_S and v_T of S and T as in Figure 2. The two pairs of horizontally aligned subtrees of each subtree order induce two subinstances that are solved recursively (in the example of Figure 2 these are $\langle S_1, T_1\rangle$ and $\langle S_2, T_2\rangle$). Each such order gives rise to a certain number of crossings at that level of the recursion (called *current-level* crossings), which is added to the number of crossings of both recursively solved subproblems (called *lower-level* crossings). Each current-level crossing has the property that it can be removed by swapping the subtrees of v_S or v_T. For example, the crossing depicted in Figure 2 can be removed by swapping the subtrees of v_S and placing S_2 above S_1. Of course such a swap generally introduces other current-level crossings. The minimum of the four possibilities is returned to the previous level of the recursion.

The two subproblems that arise from each recursive split are not independent. Nevertheless, they are treated independently by the algorithm. That is the point that introduces an error with respect to the actual number of crossings, which, for the case of *complete* binary trees, can be bounded by the number of crossings in an optimal solution [2]. For complete binary trees the recursive algorithm thus yields a 2-approximation. Obviously, the depth of the recursion

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

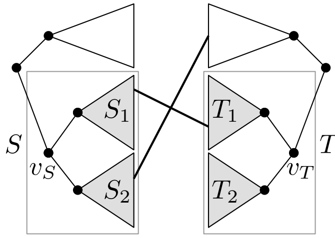
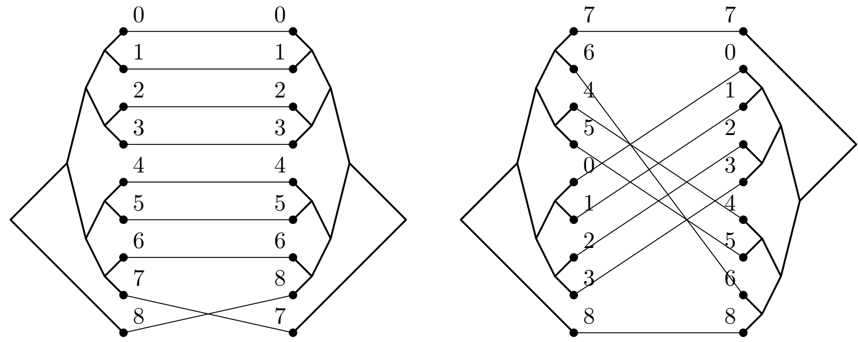


Figure 2: A subinstance $\langle S, T \rangle$ with a current-level crossing.



(a) optimal layout: 1 crossing **(b)** heuristic layout: 14 crossings

Figure 3: Example of a binary tree for which the algorithm of Buchin et al. [2] performs badly.

equals the minimum height h of the two trees. The recursion tree is of size $O(8^h)$ since each instance starts eight recursive calls (two for each of the four subtree arrangements). The computation of all current-level crossings is done in $O(4^h n)$ time, resulting in a total running time of $O(8^h + 4^h n)$. For complete trees with $h = \log n$ this resolves to $O(n^3)$ time.

In applications most binary TL instances do not consist of *complete* binary trees. The above recursive algorithm can be applied to any pair of binary trees as a heuristic but an approximation guarantee cannot be given any more. Under the widely-accepted Unique Games Conjecture a constant-factor approximation does not even exist for general binary trees [2]. Moreover, the running time grows exponentially with the height of the trees.

Still, there is room for improvements of running time and solution quality for arbitrary tanglegrams. The original algorithm always divides an instance into an upper and a lower subinstance, that is, the two problems $\langle S_1, T_1 \rangle$ and $\langle S_2, T_2 \rangle$ in the example of Figure 2. For unbalanced trees this may lead to an unnecessarily high number of ignored crossings as Figure 3 shows. The original algorithm aligns the leaves (nodes 7 and 8) attached directly to the roots since this causes no current-level crossings. All 14 crossings in Figure 3b are crossings that the algorithm does not take into account.

A small modification of the algorithm weakens this effect (and yields the optimum solution in the given example). Instead of defining a subinstance by a pair of horizontally aligned subtrees we match the four subtrees of v_S and v_T such that the number of inter-tree edges *within* each of the two subinstances is larger than the number of edges *between* one subinstance and the other. In this way less edges are disregarded in the course of the algorithm. This not only improves the performance but it also means that each subtree of S has a fixed partner in T for all branches of the recursion. Hence we can precompute in $O(n^2 h)$ time all required current-level crossings for constant-time lookup. The worst-case instance for this recursive algorithm is a pair of two caterpillar trees with linear height. In that case the running time of the recursion follows the recurrence $T(n) \leq 4T(n-1) + O(n)$ which resolves to $T(n) \in O(4^n)$. Hence the worst-case running time for arbitrary trees is $O(4^h + n^2 h)$. Note that for complete binary trees the running time is still $O(n^3)$ as for the original algorithm of Buchin et al. [2]; their analysis of the approximation factor, however, does not carry over.

In order to speed up the algorithm in practice we additionally make use of a branch-and-bound technique in order to prune large parts of the search tree as early as possible. For a subinstance $\langle S, T \rangle$ with roots v_S and v_T we first consider the arrangement of the subtrees of v_S and v_T that yields the lowest number of accumulated current-level crossings and recurse. Once the leaf level is reached this gives us an initial upper bound on the number of crossings. Now at each node of the search tree we can immediately prune the subtrees corresponding to arrangements of the current subinstance whose accumulated current-level crossings exceed this upper bound. The rest of the search tree is examined further, and each time a better solution is found, the upper bound is updated accordingly.

3.2 Hierarchy Sort The algorithm of Holten and van Wijk [9] performs a number of collapse-and-expand cycles on both trees of the binary tanglegram. During each step of these cycles, the well-known barycentric method of Sugiyama et al. [17] for 1SCM is used by successively fixing one tree, optimizing the leaf order of the other, and then changing the trees' roles until no

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

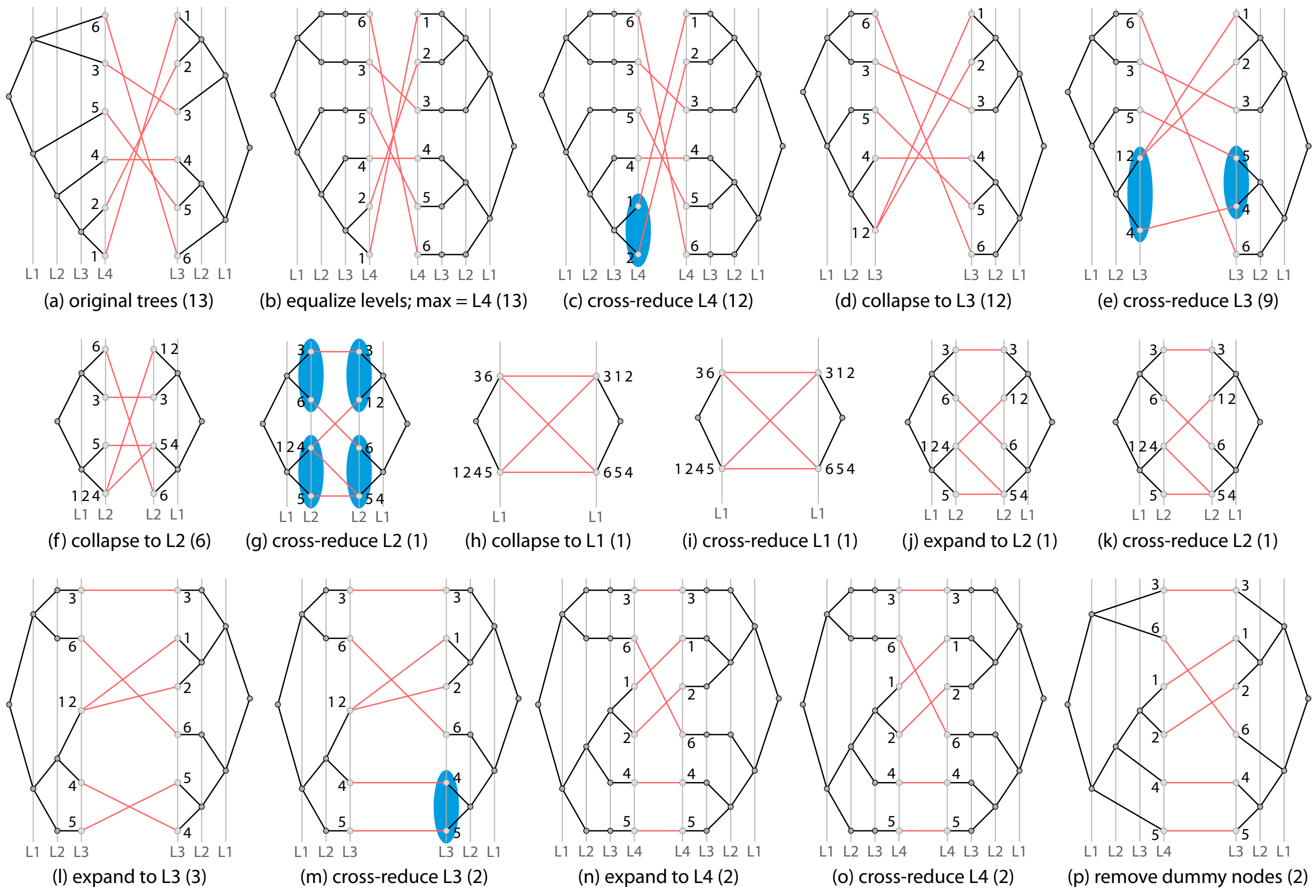


Figure 4: Step-by-step crossing reduction using the hierarchy sorting algorithm. Nodes that are swapped during crossing reduction are encircled. The numbers of inter-tree edge crossings after each step are given in parentheses.

further crossing reduction is possible.

In a first step, the two trees are augmented with dummy nodes of degree 2 below the original leaves such that both trees have the same height and all leaves are on the lowest level. Starting at the lowest level, the barycenter method is applied to both trees in turn until the leaf order gets stable. Since the leaf order is restricted by the tree topology we only need to consider sibling nodes whose parent lies one level above. For such pairs of nodes we compute the barycenter of their neighbors in the other tree and swap them if the order of the two barycenters is reversed. In the next step we collapse the lowest level of both trees replacing each inter-tree edge between two leaves by the corresponding edge between their parents. Thus for higher levels multiple inter-tree edges can be incident to a single node. Barycentric crossing reduction and collapsing are alternated until the root of both trees is reached. Now the collapsing phase is replaced by an expansion phase that adds the next level to both trees. The initial leaf order of newly expanded subtrees remains as in the corresponding previous collapsing phase and then barycentric crossing reduction is performed on that level before expanding the next level. Once the lowest level is reached a collapse-and-expand cycle is completed. The collapse-and-expand cycles are repeated until no further improvement is made. Figure 4 illustrates the algorithm step-by-step.

A simple variant of the above procedure is to introduce integer weights on the edges during collapsing such that the edge weight corresponds to the number of original edges that are represented. A further obvious variant is to reduce crossings at each level based on which configuration actually minimizes the resulting number of crossings rather than using the barycenter method. It turned out, however, that this variant performs worse than the weighted and unweighted barycenter heuristics.

The asymptotic running time of this algorithm depends of course on the number N of collapse-and-expand cycles and the maximum number N' of executions of the linear-time barycentric heuristic on each

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

level. In our experiments (see Section 4) it turned out that in all instances we had $N \leq 2$ and $N' \leq 4$ for the original heuristic. The weighted variant, on the other hand, occasionally got caught in an infinite loop of crossing reductions in one level of the algorithm, which needed to be aborted. Under the condition that both N and N' are constants, hierarchy sort runs in $O(n \cdot H)$ time, where H is the maximum height of the two trees. In the case of complete trees $H = \log n$, and the running time is $O(n \log n)$.

3.3 Iterated One-Tree Crossing Minimization One-tree crossing minimization (1TCM) is the one-sided version of TL; one tree is fixed and the other is laid out optimally. Fernau et al. [5] have shown how to solve this problem in $O(n \log^2 n)$ time for general binary trees. We denote the tree to be optimized by T. The main observation is that the crossing behavior of any two inter-tree edges depends only on the swapping decision taken at their lowest common ancestor node v in T. Hence we can apply a divide-and-conquer strategy to recursively determine the optimal layout of the two subtrees of v and then arrange them at v such that the number of crossings caused by v is minimum. In our current implementation the divide-and-conquer algorithm still has a running time of $O(n^3)$. Dwyer and Schreiber [3] suggested to apply an algorithm for 1TCM in turn to the two given trees until a local optimum for TL is found.

3.4 Integer Linear Program We now give a simple formulation of binary TL as an integer linear program (ILP). Let S° and T° denote the sets of inner nodes of the given binary trees S and T, respectively. We introduce a binary variable x_u for each node $u \in S^\circ \cup T^\circ$. If $x_u = 1$, the two subtrees of u change their order with respect to the input drawing, otherwise the order of the input drawing is kept. Any assignment of these variables corresponds to a tanglegram layout. Let ab and cd be two inter-tree edges with $a, c \in S$ and $b, d \in T$. Let $v \in S$ and $w \in T$ be the lowest common ancestors of the leaves a and c, and of b and d, respectively. We distinguish two cases: If ab and cd cross in the input layout, they cross in any layout that either swaps the subtrees of both v and w or that does not swap either of them. In other words, there is a crossing if and only if $x_v = x_w$. Similarly, if ab and cd do not cross in the input layout, they cross in any layout that swaps the subtrees of exactly one of v or w, or, equivalently, ab and cd cross if and only if $x_v \neq x_w$.

Now for a pair $(v, w) \in S^\circ \times T^\circ$, let $k^\times_{vw}$ ($k^=_{vw}$) be the number of edge pairs that have v and w as their lowest common ancestors and that (do not) cross in the initial layout. Note that $k^\times_{vw}$ and $k^=_{vw}$ are constants (from the ILP point of view). A table with their values for any choice of $(v, w) \in S^\circ \times T^\circ$ can be precomputed in $O(n^2)$ time as follows. We initialize all values with 0 and preprocess the two trees S and T in linear time in order to support constant-time lowest-common ancestor queries [6]. Then we determine for each pair of inter-tree edges their lowest common ancestors in S and T and increment the corresponding table entry depending on which two configurations yield the crossing. This takes $O(n^2)$ total time for all edge pairs.

Finally, we would like to define a variable y_{vw} for each pair $(v, w) \in S^\circ \times T^\circ$ that equals 1 if $x_v \neq x_w$ and 0 otherwise. This can be achieved with four linear constraints and yields the following ILP formulation for binary TL:

$$\begin{aligned}
&\text{Minimize} && \sum_{v \in S^\circ,\, w \in T^\circ} \left[y_{vw} \cdot k^=_{vw} \;+\; (1 - y_{vw}) \cdot k^\times_{vw} \right] \\
&\text{subject to} && \left.\begin{array}{rcl} y_{vw} & \leq & 2 - x_v - x_w \\ y_{vw} & \leq & x_v + x_w \\ y_{vw} & \geq & x_v - x_w \\ y_{vw} & \geq & x_w - x_v \end{array}\right\} \quad \forall v \in S^\circ,\, w \in T^\circ.
\end{aligned}$$

3.5 Exact Branch-and-Bound Algorithm The branch-and-bound idea used in the implementation of the recursive splitting algorithm of Section 3.1 can also be applied to compute optimal solutions. The main ingredients for speeding up the search are: (i) ordering the nodes in the search tree according to the impact of the swapping decisions to quickly find good solutions and (ii) using lower bounds that are as tight as possible to cut off large parts of the search tree as early as possible.

We preprocess the given binary TL instance $\langle S, T \rangle$ by computing for any pair $(v, w) \in S^\circ \times T^\circ$, the numbers $k^\times_{vw}$ and $k^=_{vw}$ of edge crossings that are induced by the swapping decisions at v and w. This takes $O(n^2)$ time as described in the previous subsection. We further store an *interaction counter* $\text{ic}(v)$ for each node $v \in S^\circ \cup T^\circ$ that contains the number of inner nodes w in the opposite tree for which $|k^\times_{vw} - k^=_{vw}| > 0$ (otherwise the swapping decisions for v and w are independent). Then we construct and traverse the search tree starting at the node whose interaction counter has the largest value. At each subsequent step we consider the next unvisited node $v \in S^\circ \cup T^\circ$ that has the largest difference between the number of crossings induced by swapping and by not swapping its subtrees given all the previous decisions in the search tree. If $\text{ic}(v) = 0$, we simply assign the better choice for v since no further decisions depend on v. Otherwise we compute for both swap options at v a lower bound on the number of crossings arising from all subsequent nodes of $S \cup T$. This is done

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

by summing up in linear time the respective minima of the two possible crossing numbers at each node given the decisions taken so far in the traversal of the search tree, including v. If the sum of this lower bound on future crossings and the number of current crossings is greater or equal to the current best solution, we can safely cut off the current branch of the search tree. Otherwise we update the number of induced crossings for the remaining nodes accordingly (using the precomputed crossing numbers), decrease the interaction counters for the remaining nodes that interact with v by one, and go further down the search tree. Once all nodes of the search tree have been visited or cut off, we return the best solution found.

The running time of the algorithm is $O(n^2 + n \cdot 2^{2n})$ in the worst case since we spend $O(n)$ time per node of the search tree. After the $O(n^2)$-time preprocessing phase, however, we reach the first leaf of the search tree—and thus a valid tanglegram layout—as soon as a first assignment of the layout of the $O(n)$ inner nodes of $\langle S, T \rangle$ has been made. This takes only $O(n^2)$ time in total. We include this greedily found first solution of the branch-and-bound algorithm as an additional fast heuristic in our evaluation.

4 Experimental Results

We have implemented all algorithms described in the previous section in Java 1.6. We have executed them on an AMD Opteron 2218 2.6 GHz system with 8 GB RAM under SuSE Linux 10.3. For solving the ILP we used the Java API of the commercial mathematical programming software CPLEX 11.1. The primary goal of our study is to evaluate which of the proposed algorithms best solves binary TL for real-world instances. The most important criterion is thus the performance ratio with respect to the optimal solution in terms of the number of crossings. A secondary goal is to identify algorithms that are fast enough to be used interactively in a tanglegram visualization tool. In the following we first introduce our test data and then evaluate performance and running times of the heuristics.

4.1 Data We generated four sets (A–D) of random tanglegrams. Set A contains 100 pairs of complete binary n-leaf trees with random leaf orders for each $n \in \{16, 32, 64, 128, 256, 512\}$. In set B we simulate tree mutations by starting with two identical complete binary trees and then randomly swapping the positions of up to 20% of the leaves of one tree. This is done as follows: we first pick a leaf uniformly at random and then iteratively climb up the tree with probability 0.75 in each step. From the node thus reached we climb back down and choose its left or right child with equal probability until we reach another leaf. This leaf and the leaf picked in the beginning are swapped. Thus the probability of two leaves being swapped decreases with their distance in the tree. Set C contains 100 pairs of general binary n-leaf trees for each $n \in \{20, 30, \ldots, 300\}$. The trees are constructed from a set of nodes, initially containing the n leaves, by iteratively joining two random nodes in a new parent node that replaces its children in the set. This process generates trees that resemble phylogenetic trees or clustering dendrograms. It mimics the hierarchical construction of these trees, which iteratively joins the two closest clusters or set of species. Set D is similar to set C but again in each tanglegram the second tree is a mutation of the first tree, where up to 10% of the leaves can swap positions as done in set B and up to 25% of the subtrees can reattach to another edge. The edge for attaching the subtree is selected in a random walk starting at the subtree's old position. The walk continues with probability 0.75 and picks the left or right edge with equal probability. Trees in this set are of interest since real-world tanglegrams often consist of two related and rather similar trees.

Our real-world examples comprise three sets (E–G) of 1303 pairs of phylogenetic trees of animal gene families obtained from the TreeFam database[1] [11, 12]. The trees in set E were generated from the multiple sequence alignments provided by TreeFam using the phylogenetic tree construction software treebest[2] [11]. For each database entry the first tree was built using the maximum-likelihood algorithm PHYML [7] and the second tree using the distance-based method neighbor joining (nj) [16]. Both methods are widely used in bioinformatics and have turned out to generate trees that are closest to the manually curated trees in TreeFam [11]. Neighbor-joining depends on a distance measure that reflects the probabilities of mutations at the positions in the underlying DNA or protein sequences. Two commonly used distances are the synonymous distance (ds) and the non-synonymous distance (dn). Distance dn is more appropriate for modeling long-term evolution while ds covers more recent mutation events better. The nj-trees in set E are actually constructed by a tree-merge algorithm described by Li [11] and implemented in treebest that builds a consensus tree of the nj-trees obtained from ds and dn. It is an obvious question to compare the merged tree with its two source trees. Therefore, sets F and G contain the tanglegrams consisting of the merged nj-tree and its underlying ds- or dn-tree, respectively. All three data sets E–G share the

[1] http://www.treefam.org
[2] http://treesoft.sourceforge.net

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

fact that about 75% of the trees have less than 50 leaves and only 5% have more than 100 leaves.

The crossing numbers of our examples are depicted in Figure 5. They vary strongly: as to be expected, mutated trees (B and D) have far lower crossing numbers than random pairs of trees (A and C). The TreeFam tanglegrams in sets E and G are generally characterized by low crossing numbers—at least for $n < 200$. Only set F and the largest trees in set E have relatively high crossing numbers ranging between those in sets C and D.

4.2 Performance In the following we denote the improved recursive splitting algorithm (Section 3.1) by *rec-split++*. The hierarchy sort algorithm (Section 3.2) is abbreviated by *hierarchy-sort* and its variant using weighted edges by *hierarchy-sort++*. The iterated 1TCM algorithm (Section 3.3) is denoted as *1tcm-iterated* and the heuristic that yields the first solution obtained in the exact branch-and-bound method (Section 3.5) is called *bb-1st-sol*. Let n be the size of an instance, that is, the number of leaves per tree.

To each tanglegram we applied the five heuristics, the ILP, and the exact branch-and-bound algorithm *bb-exact*. We recorded the number k_i of crossings in the output, i being one of the heuristics in {*rec-split++*, *hierarchy-sort*, *hierarchy-sort++*, *1tcm-iterated*, *bb-1st-sol*}. We only recorded results that were obtained within at most one minute wall clock time. For each tanglegram we computed the performance ratio $(k_i + 1)/(k + 1)$. The crossing number k was obtained from one of the two exact algorithms. Note that we add one to both numerator and denominator such that the ratio is also defined for crossing-free instances.

We have also implemented the original recursive splitting algorithm *rec-split*, but we chose not to include it in our detailed evaluation for several reasons. First of all, *rec-split* is designed only for complete binary trees while *rec-split++* is adapted to deal with unbalanced trees as well. Hence *rec-split* performs badly on general binary tanglegrams, the focus of our evaluation. Moreover, for complete binary tanglegrams, both variants perform almost identically. Lastly, *rec-split* is far slower than *rec-split++*: complete binary tanglegrams with $n \geq 512$ could not be solved within one minute wall time by *rec-split*. For general binary trees, this timeout was reached already for several instances with $n \approx 80$.

In the subsequent discussion we refer to the performance ratios shown in Figure 6. A first inspection of the plots immediately reveals that there is a clear method of choice for all our examples that not only outperforms the other heuristics but even achieves average performance ratios hardly deviating from the optimum: *bb-1st-sol*. This comes as a surprise since *bb-1st-sol* is merely a byproduct of our exact branch-and-bound algorithm while the other heuristics were explicitly designed to obtain good solutions efficiently.

Next, we examine the results for the different sets of tanglegrams in more detail. We start with the complete binary trees in sets A and B. Set A with random pairs of trees shows that *hierarchy-sort* performs worst and spreads over a relatively large range of values. On the other hand the variant *hierarchy-sort++* is among the best heuristics apart from *bb-1st-sol*. For $n \geq 64$ *rec-split++* and *1tcm-iterated* catch up with *bb-1st-sol* and *hierarchy-sort++* and also achieve performance ratios between 1 and 1.1.

For set B containing mutated trees that are more similar to each other, the picture changes. Algorithms *bb-1st-sol* and *rec-split++* outclass the the other three methods with average performance ratios below 1.01 and many optimal solutions while the other methods range between 1.6 and 4. In terms of outliers *rec-split++* is slightly preferable to *bb-1st-sol*. Comparing sets A and B it is noteworthy that *rec-split++* and *bb-1st-sol* perform equally well on random and mutated trees, while the remaining methods are susceptible to the similarity and consequently the crossing number of the two trees, see Figure 5. Recall that *rec-split++* is based on the 2-approximation algorithm for complete binary trees of Buchin et al. [2]. Although we have not been able to prove *rec-split++* to be a 2-approximation, too, the performance ratio of *rec-split++* in the experiments is far better than 2 even in the worst cases.

For the general binary trees in sets C and D the five heuristics have a similar ranking as for the sets A and B. Algorithm *bb-1st-sol* remains the best method with average performance ratios below 1.01 and is even more clearly ahead of the remaining algorithms. For random pairs of trees (set C) the three methods *rec-split++*, *hierarchy-sort++*, and *1tcm-iterated* show a similar performance, which is on average below 1.1. The worst performance is again obtained by *hierarchy-sort*.

For mutated trees (set D) *bb-1st-sol* is again almost optimal but this time *rec-split++* performs a lot better than the remaining competitors, which is similar to the situation for set B. While *rec-split++* shows performance ratios between 1 and 2 even for the third quartile, *hierarchy-sort++* and *1tcm-iterated* lie on average between 2 and 4. The original method *hierarchy-sort* even reaches average ratios close to 7. It should be noted that outliers for all algorithms except *bb-1st-sol* reach values between 10 and 100. Furthermore, for random pairs of trees the completeness does not seem to affect the quality of the solutions since the results for sets A and C are very similar. For mutated trees in set D, *rec-split++* becomes inferior to *bb-1st-sol* unlike the results

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

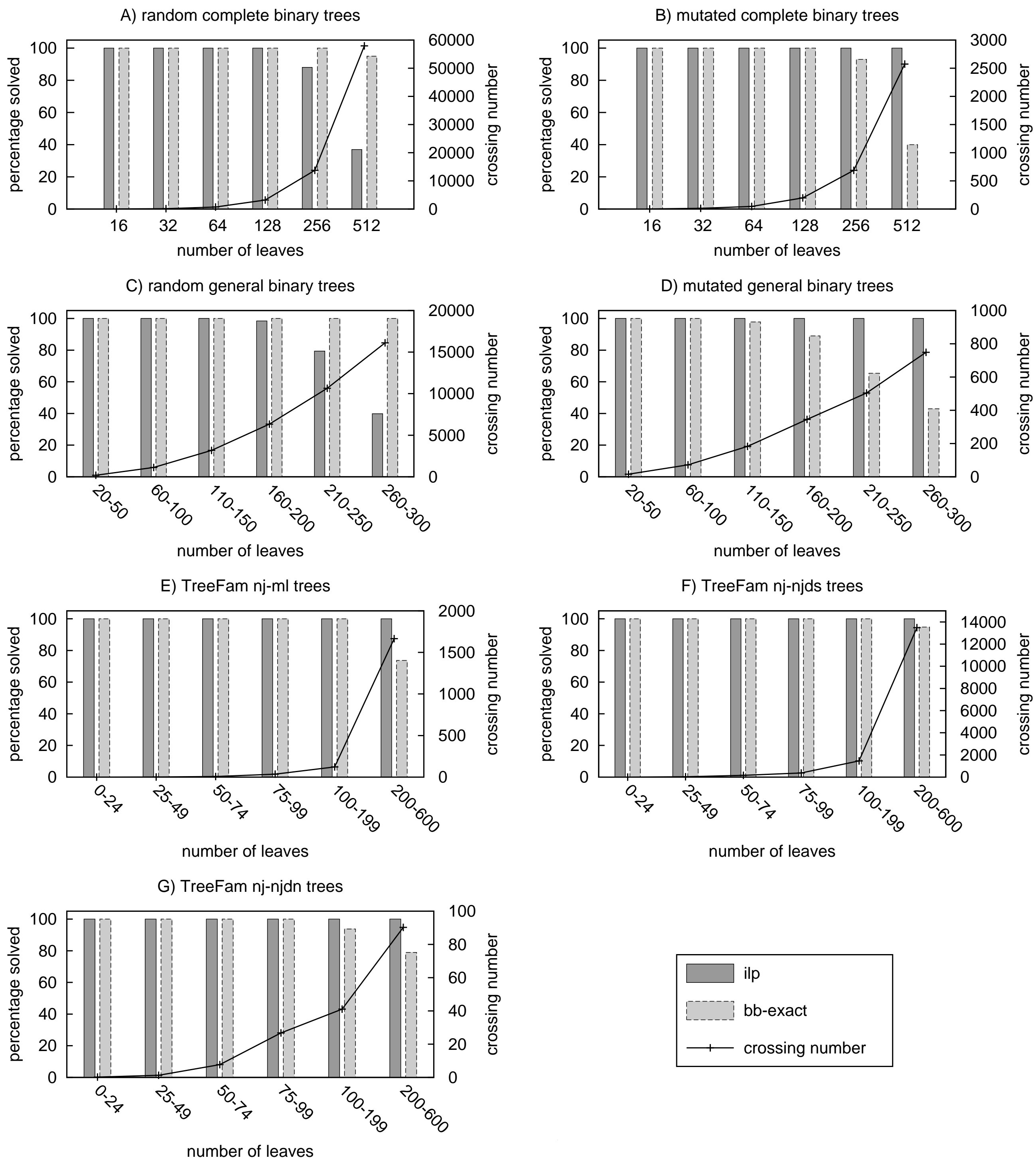


Figure 5: Percentage of solved instances for the exact algorithms *ilp* and *bb-exact* (left axes) and average crossing numbers (right axes) of random (A–D) and real-world (E–G) binary tanglegrams. Note the different scales on the right axes.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

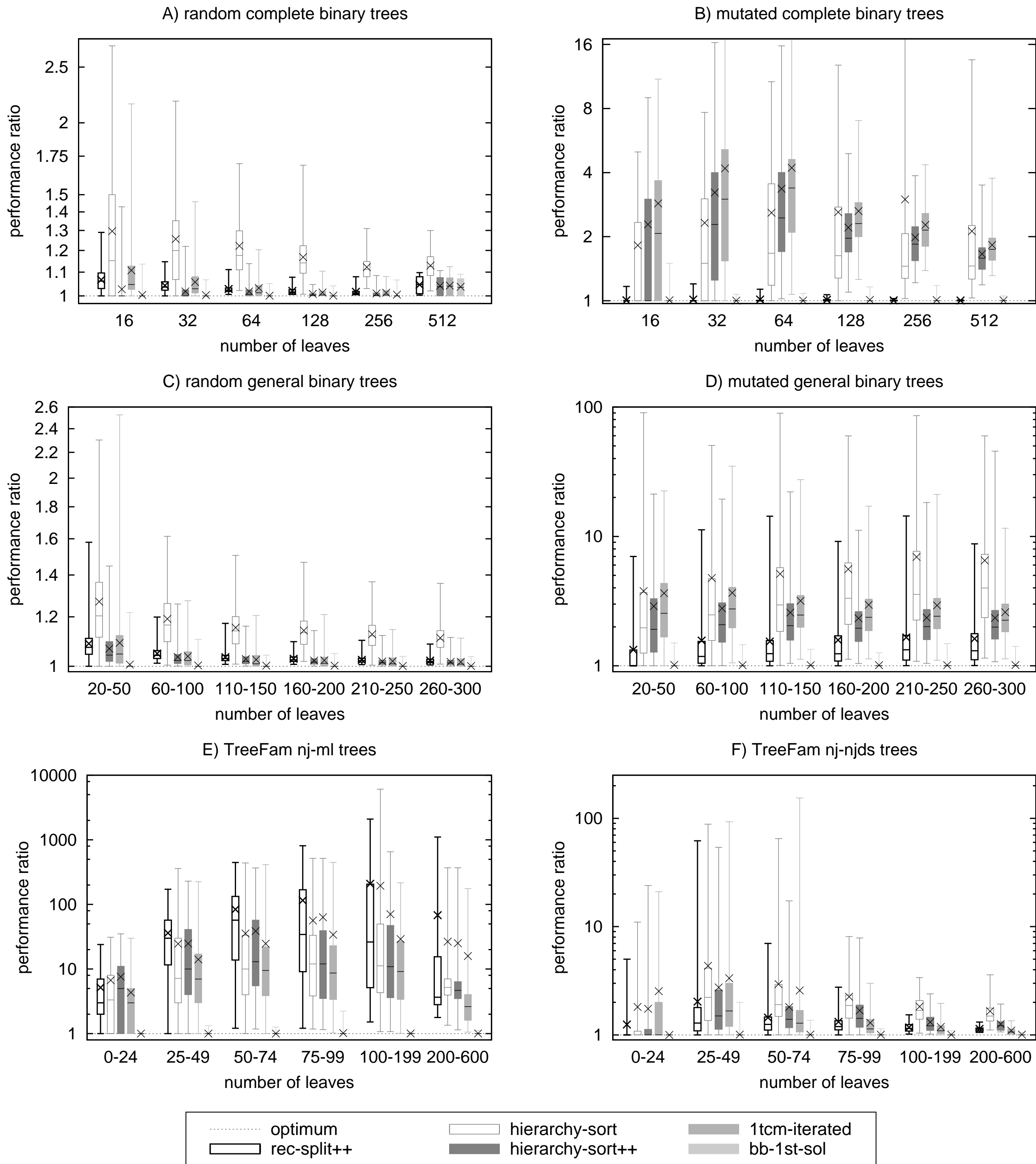


Figure 6: Performance ratios of the five algorithms *rec-split++*, *hierarchy-sort*, *hierarchy-sort++*, *1tcm-iterated*, and *bb-1st-sol* for random (A–D) and real-world (E–F) binary tanglegrams. The boxplots show median (−), arithmetic mean (×), first and third quartile, minimum and maximum.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

for set B. Also the performance ratios of the algorithms on the whole (except *bb-1st-sol*) spread a lot more in set D than in set B.

The relative performance of the heuristics on the real-world data partially mirrors our observations from sets C and D but also exhibits different effects. Note that due to space constraints we omit the plot of the performance ratios for set G. The results for set G are, however, similar to those of set E since both these sets contain rather similar trees. In analyzing the results for sets E–G recall that about 95% of the trees have $n \leq 100$ leaves and that crossing numbers vary a lot, see Figure 5. In the sets E and G, the crossing numbers for $n \leq 100$ are roughly the same, ranging between 0 and 35 on average. For trees of size $n > 100$, the crossing numbers in set E increase drastically; those in set G remain small. On the other hand, the crossing numbers in set F are higher by a factor of at least 10 in comparison to sets E and G.

The results for all three sets have in common that, as before, *bb-1st-sol* attains by far the best performance ratios and even finds optimal solutions for over 75% of the instances. Moreover, the remaining heuristics have severe problems with outliers that reach inacceptable ratios between 100 and 1000, in some cases even worse. Note, however, that the ratio $(k_i+1)/(k+1)$ equals (almost) the absolute number of crossings k_i for instances with $k = 0$. In set E the second best method is *1tcm-iterated*, followed by the two hierarchy-sort variants. Interestingly, *hierarchy-sort++* is no longer preferable to *hierarchy-sort*. The algorithm *rec-split++* performs worst on set E. The order of the algorithms is thus quite different from that on the random set D although size and crossing number of the trees is similar.

Set F with less similar trees gives results that are comparable to set C in terms of the relative order of the algorithms. The *hierarchy-sort++* heuristic is worst, at least for $n \geq 25$. For instances with $n \leq 100$, *rec-split++* ranks second after *bb-1st-sol*. Finally, *hierarchy-sort++* performs better than *1tcm-iterated* for small instances, while *1tcm-iterated* beats *hierarchy-sort++* on the large instances. In absolute numbers, however, outliers in set F are a lot worse than in set C.

4.3 Running Time Although the number of crossings is the main aspect for assessing the quality of TL algorithms, their running time is also an important criterion—especially if the layouts are to be produced interactively. Figure 7 shows plots of the median running times of our five heuristics as well as of the integer program *ilp* (Section 3.4) and the exact branch-and-bound algorithm *bb-exact* (Section 3.5). In our experiments there was a timeout after one minute wall clock time for all algorithms. Note that the running times summarize regularly terminated runs and those aborted after one minute. Also note the different scales on the x-axes. The main question is whether *bb-1st-sol*, whose performance ratio is far better than the other four heuristics and which even finds optimal solutions in most of the cases, is fast enough to be used in practice. Moreover, it is of interest whether we can afford to compute optimal solutions for typical input sizes.

Let's first consider the running time of *bb-1st-sol*. For small instances *bb-1st-sol* is among the fastest methods with running times between 0.001 and 0.01 seconds. For larger instances the running times grow to values between 0.1 and 0.25 seconds, placing *bb-1st-sol* in the mid-range of the other heuristics. Nonetheless, even the largest instances are solved in at most half a second. Further note that *bb-1st-sol* has a worst-case running time of $O(n^2)$ although *bb-exact*, from which it is derived, has an exponential worst-case running time. From the remaining heuristics, *1tcm-iterated* is fastest for small instances, while *rec-split++* and the two hierarchy-sort variants are faster for larger instances. Recall that our implementation of *1tcm-iterated* runs in $O(n^3)$ time while a faster $O(n \log^2 n)$-time implementation is possible [5]. For complete binary trees, *rec-split++* is not only a 2-approximation but it is also the fastest algorithm for $n \geq 128$. An interesting observation is that *hierarchy-sort++* takes less time for $n = 512$ than for $n = 256$ on sets A and B. Summarizing the running times of the heuristics it turned out in our experiments that all of them are fast enough to be applied interactively in a tanglegram visualization tool. In particular, this is the case for *bb-1st-sol*, the heuristic with the best performance.

Considering the exact algorithms, note that *ilp* is slower than the five heuristics by a factor between 10 and 100. Still, it is remarkable that it succeeds to find optimal solutions in less than 10 seconds for mutated random trees (sets B and D) and the TreeFam instances. Only random pairs of trees (sets A and C) with many crossings in the optimal solution are challenging and running times reached the timeout with increasing frequency as Figure 5 shows. The running times of *bb-exact* behave oppositely. While *bb-exact* is as fast as the heuristics for small values of n, its running time increases quickly for trees that are relatively similar to each other (sets B and D), where *bb-exact* gets slower than *ilp*. Hence the upper bounds used to prune subtrees of the search tree in *bb-exact* seem most efficient for instances of unrelated trees with large crossing numbers. For the largest of our instances *bb-exact* does no longer find all optimal solutions within one minute, most notably for sets B and D (see Figure 5).

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

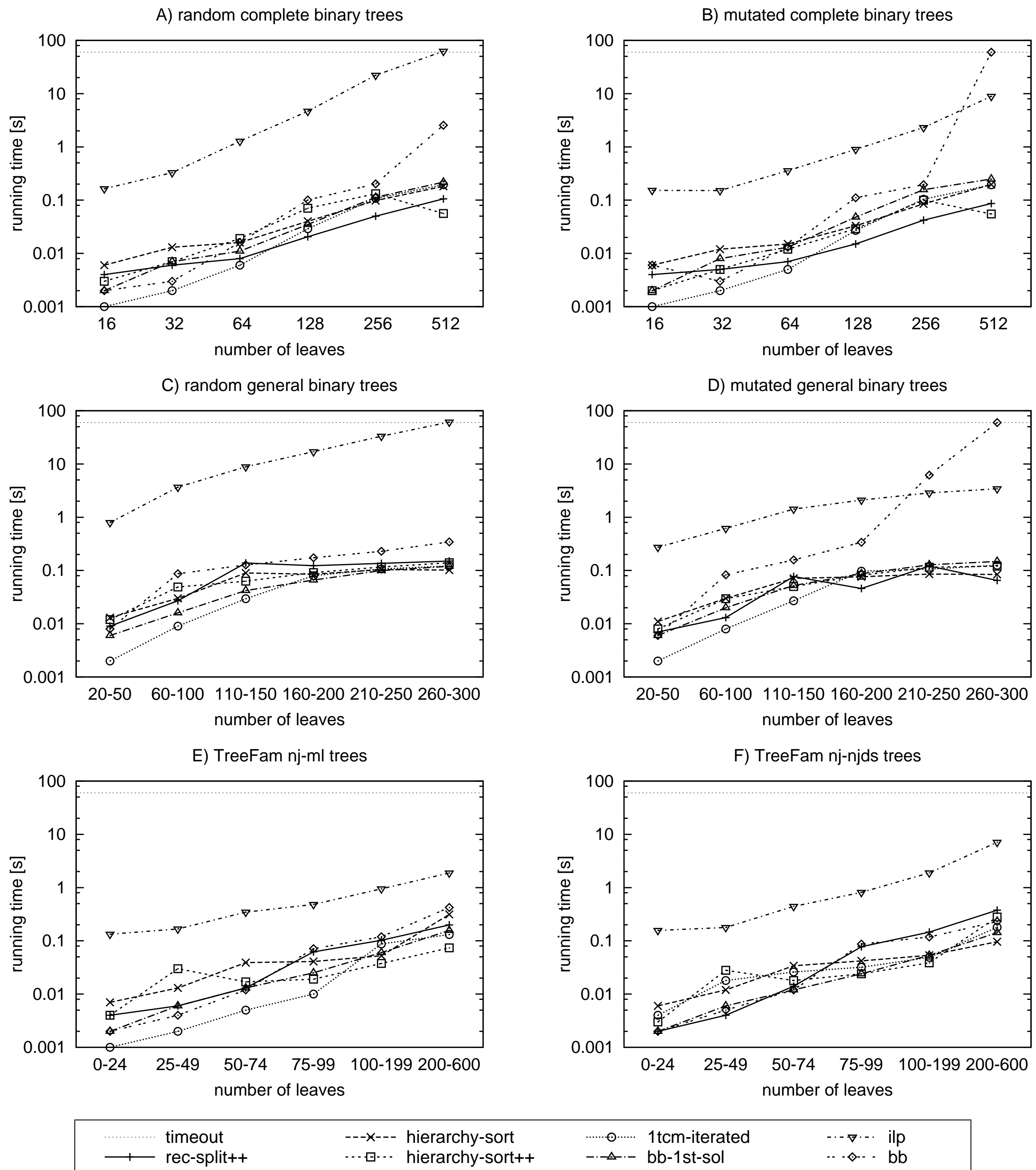


Figure 7: Median running times of the algorithms *rec-split++*, *hierarchy-sort*, *hierarchy-sort++*, *1tcm-iterated*, *bb-1st-sol*, *ilp*, and *bb-exact* (in seconds) for random (A–D) and real-world (E–F) binary tanglegrams.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

5 Conclusions

The experimental evaluation clearly indicates that *bb-1st-sol*, the first solution obtained by our exact branch-and-bound algorithm, is superior to all other heuristics and thus the method of choice for arbitrary binary tanglegrams. It found optimal solutions in more than 82% of our examples; the structural reason for this astonishing behavior is an interesting question in its own. The worst-case performance of *bb-1st-sol* observed in the more than 10,000 examples was 2.24. With a running time of $O(n^2)$ *bb-1st-sol* also turned out to be fast enough even for the largest tanglegrams of size up to 600 leaves, which took less than half a second to compute. In practice it might be a good idea to continue running *bb-exact* for a pre-specified time after the first solution has been found in order to find an even better (or the optimal) solution. The ILP is faster for large instances than *bb-exact* (if crossing numbers are not too high) and its running time is less susceptible to outliers. Its disadvantage is, however, that our implementation requires a license for a commercial ILP solver. Note that real-world tanglegrams often have a crossing-free layout. By construction, the underlying algorithm *bb-exact* guarantees to find a crossing-free layout as the first solution if it exists. Thus *bb-1st-sol* is optimal in that case. For the special case of complete binary trees, the 2-approximation algorithm *rec-split++* is faster than *bb-1st-sol* and achieves a similar performance, which makes it a good alternative for that case.

6 Open problems

At least two ways of generalizing binary TL are interesting for applications. In software analysis, trees represent package-class-method hierarchies or the decomposition of a project into layers, units, and modules. Such trees are usually not binary. Since tanglegrams with a small crossing number seem to occur often in practice, it would be desirable to have an FPT algorithm for general (non-binary) trees that is simpler and faster than the algorithm of Fernau et al. [5].

A different way of generalizing TL is of interest in our main application, in phylogenetic trees. There, sometimes a leaf of one tree corresponds to two or more leaves of the other tree. This can be formalized by requiring that the leaf sets of the two trees induce a collection of stars. The algorithms evaluated in this paper all generalize to this setting with minor adjustments. Would this still allow for constant-factor approximations in the case of complete binary trees?

References

[1] P. Bertolazzi, G. Di Battista, C. Mannino, and R. Tamassia. Optimal upward planarity testing of single-source digraphs. *SIAM J. Comput.*, 27(1):132–169, 1998.

[2] K. Buchin, M. Buchin, J. Byrka, M. Nöllenburg, Y. Okamoto, R. I. Silveira, and A. Wolff. Drawing (complete) binary tanglegrams: Hardness, approximation, fixed-parameter tractability. In I. G. Tollis and M. Patrignani, editors, *Proc. 16th Internat. Sympos. Graph Drawing (GD'08)*, volume 5417 of *Lecture Notes Comput. Sci.*, pages 324–335. Springer-Verlag, 2009. Full version available at http://arxiv.org/abs/0806.0920.

[3] T. Dwyer and F. Schreiber. Optimal leaf ordering for two and a half dimensional phylogenetic tree visualization. In N. Churcher and C. Churcher, editors, *Proc. Australasian Sympos. Inform. Visual. (InVis.au'04)*, volume 35 of *CRPIT*, pages 109–115. Australian Computer Society, 2004.

[4] P. Eades and N. Wormald. Edge crossings in drawings of bipartite graphs. *Algorithmica*, 10:379–403, 1994.

[5] H. Fernau, M. Kaufmann, and M. Poths. Comparing trees via crossing minimization. In R. Ramanujam and S. Sen, editors, *Proc. 25th Intern. Conf. Found. Softw. Techn. Theoret. Comput. Sci. (FSTTCS'05)*, volume 3821 of *Lecture Notes Comput. Sci.*, pages 457–469, 2005.

[6] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. In *Proc. 15th Ann. ACM Symp. Theory Comput. (STOC'83)*, pages 246–251, 1983.

[7] S. Guindon and O. Gascuel. A simple, fast, and accurate algorithm to estimate large phylogenies by maximum likelihood. *Systematic Biology*, 52(5):696–704, 2003.

[8] M. S. Hafner, P. D. Sudman, F. X. Villablanca, T. A. Spradling, J. W. Demastes, and S. A. Nadler. Disparate rates of molecular evolution in cospeciating hosts and parasites. *Science*, 265:1087–1090, 1994.

[9] D. Holten and J. J. van Wijk. Visual comparison of hierarchically organized data. In *Proc. 10th Eurographics/IEEE-VGTC Sympos. Visualization (EuroVis'08)*, pages 759–766, 2008.

[10] M. Jünger and P. Mutzel. 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. *J. Graph Algorithms Appl.*, 1(1):1–25, 1997.

[11] H. Li. *Constructing the TreeFam Database*. PhD thesis, The Institute of Theoretical Physics, Chinese Academy of Science, 2006.

[12] H. Li, A. Coghlan, J. Ruan, L. J. Coin, J.-K. Hériché, L. Osmotherly, R. Li, T. Liu, Z. Zhang, L. Bolund, G. K.-S. Wong, W. Zheng, P. Dehal, J. Wang, and R. Durbin. TreeFam: a curated database of phylogenetic trees of animal gene families. *Nucleic Acids Research*, 34:D572–D580, 2006.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

[13] A. Lozano, R. Y. Pinter, O. Rokhlenko, G. Valiente, and M. Ziv-Ukelson. Seeded tree alignment and planar tanglegram layout. In R. Giancarlo and S. Hannenhalli, editors, *Proc. 7th Internat. Workshop Algorithms Bioinformatics (WABI'07)*, volume 4645 of *Lecture Notes Comput. Sci.*, pages 98–110. Springer-Verlag, 2007.

[14] X. Muñoz, W. Unger, and I. Vrt'o. One sided crossing minimization is NP-hard for sparse graphs. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Proc. 9th Internat. Sympos. Graph Drawing (GD'01)*, volume 2265 of *Lecture Notes Comput. Sci.*, pages 115–123. Springer-Verlag, 2002.

[15] R. D. M. Page, editor. *Tangled Trees: Phylogeny, Cospeciation, and Coevolution*. University of Chicago Press, 2002.

[16] N. Saitou and M. Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4:406–425, 1987.

[17] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, 1981.

[18] W. N. W. Zainon and P. Calder. Visualising phylogenetic trees. In W. Piekarski, editor, *Proc. 7th Austalasian User Interface Conf. (AUIC'06)*, volume 50 of *CRPIT*, pages 145–152. Australian Comput. Soc., 2006.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.