The Computer Journal Advance Access published June 28, 2016

doi:10.1093/comjnl/bxw018

Complete Balancing via Rotation

Fabrizio Luccio[1], Bernard Mans[2], Luke Mathieson[2*], and Linda Pagli[1]

[1]*Dipartimento di Informatica, Università di Pisa, Largo Bruno Pontecorvo, 3, 56127 Pisa, Italy*
[2]*Department of Computing, Macquarie University, Sydney, NSW 2109, Australia*
[*]*Corresponding author: luke.mathieson@gmail.com*

Trees are a fundamental structure in algorithmics. In this paper, we study the transformation of an arbitrary binary tree S with n vertices into a completely balanced tree T via *rotations*, a widely studied elementary tree operation. Combining concepts on rotation distance and data structures, we give a basic algorithm that performs the transformation in $\Theta(n)$ time and $\Theta(1)$ space, making at most $2n - 2\log_2 n$ rotations and improving on known previous results. The algorithm is then improved, exploiting particular properties of S. Finally, we show tighter upper bounds and obtain a close lower bound on the rotation distance between a zig-zag tree and a completely balanced tree. We also find the exact rotation distance of a particular almost balanced tree to a completely balanced tree, and thus show that their distance is quite large despite the similarity of the two trees.

Keywords: rotation distance; tree balancing; complete binary tree; design of algorithms

Received 7 October 2015; revised 9 February 2016
Handling editor: Daniel Paulusma

1. INTRODUCTION

The *rotation distance* $d(S,T)$ is the minimal number of rotations needed to transform a given binary tree S into another binary tree T both with n vertices. The problem of computing $d(S,T)$ was introduced by Culik and Wood [1] in 1982 and has been extensively studied since then. Surprisingly, it is still not known if this problem is NP-hard.

An upper bound $d(S,T) \leq 2n - 6$ for $n \geq 11$ was given by Sleator, Tarjan and Thurston in a seminal paper [2]. More importantly, using a deep geometric argument they proved that the upper bound $2n - 6$ is sharp for (ineffectively) large values of n, and conjectured that it is sharp for any $n \geq 11$. This conjecture has been definitely settled by Pournin [3]. Mäkinen [4] showed that a slightly weaker upper bound can be obtained by an elementary procedure, and Luccio and Pagli [5] gave an alternative proof for the upper bound of Ref. [2] that makes explicit use of binary trees. No polynomial time algorithm is known to compute $d(S,T)$. Heuristic, approximate and parameterized algorithms have been given in Refs [6–11].

In this paper, we look at a new formulation of the problem where S is an arbitrary tree and the target tree T is almost balanced. That is, T has the desirable property that all the paths from the root to the leaves have a strictly logarithmic length. Transforming a binary tree into a balanced tree is a basic operation, which is important by itself and as subroutine. In fact, the two important families of *general Boolean trees* of Andersson [12] and the *scapegoat trees* of Galperin and Rivest [13] use such an operation. In both structures, the tree is allowed to be unbalanced up to a given threshold value, otherwise a global rebalancing operation is performed at the root of a subtree transforming it in an almost balanced tree.

There is an immediate recursive algorithm for transforming a sorted array into a balanced tree, which selects the central element of the array as the root and recursively builds the left and right subtrees of the root from the subarrays at the left and at the right of the central element. While this algorithm requires that the tree elements be copied in sorted order into an array and that a new tree is constructed, the problem can be solved *in-place* without additional space through rotations. As reported in Ref. [14], there is a vast literature on this subject and the best proposed algorithm is a generalization of the Stout and Warren algorithm [15] requiring $O(n)$ rotations. The exact number of rotations required by this generalization can be easily computed as $2n - \lceil \log_2 n \rceil$ in the worst case. As we shall see that this is not the best that can be achieved in terms of an exact (not asymptotic) number of rotations. As far as we are aware, none of the literature addresses the rotation distance problem to a balanced tree.

In this paper, we exploit the almost completely balanced structure of the target tree to obtain better upper bounds on the rotation distance between an arbitrary tree and a completely balanced tree using an ad-hoc transformation algorithm. In particular, we prove that this distance is bounded above by $2n - 2\lfloor \log_2 n \rfloor$, thus improving upon the above-mentioned result. We then show how the proposed transformation algorithm can be extended when subtrees of S and T have the same vertices, attaining a further reduction of the number of rotations. Finally, we show tighter upper bounds and obtain lower bounds on the rotation distance between a zig-zag tree and a completely balanced tree. We also find the exact rotation distance of a particular almost balanced tree to a completely balanced tree. This last case is interesting because, despite the similarity of these trees their distance is quite large. For this purpose, we first introduce our notations and recall some basic properties.

2. BASIC PROPERTIES OF THE ROTATION DISTANCE

Let S and T be two rooted binary trees with n vertices. The vertices of each tree can be numbered from 1 to n according to their occurrence in an in-order traversal of the tree, where the integer label assigned to each vertex is greater (respectively, smaller) than the labels of the vertices of its left (respectively, right) subtree. In the following, vertices will be identified with their labels, so an integer x may refer to one vertex in S and to the same vertex in T.

A *right rotation* on vertices x,y, where y is the left child of x (hence $y < x$), is denoted by rot(x,y) and is defined as a right rotation where y takes the place of x. Symmetrically, a *left rotation* on vertices x,y, where y is the right child of x (hence $y > x$), is also denoted by rot(x,y) and is defined as the corresponding left rotation. The application of rot(x,y) followed by rot(y,x) on the same tree leaves the tree unchanged. The rotations take constant time, since they simply consist in modifying three parameters. See Fig. 1 for an example of simple rotations. The *rotation distance* $d(S,T)$ between S and T is the minimum number of rotations needed to transform S into T. For any S and T, the rotation distance is of course symmetric, i.e. $d(S,T) = d(T,S)$.

A *complete binary tree* of height h has $n = 2^h - 1$ vertices all contained in h levels. We regard it as a special case of an *almost complete binary tree* where the number of vertices is $2^{h-1} \leq n \leq 2^h - 1$, the first $h - 1$ levels are filled with $2^{h-1} - 1$ vertices, and the remaining $n - (2^{h-1} - 1)$ vertices are in the leftmost positions of level h. Here, we consider the case where S is any binary tree and T is an almost complete binary tree. For this purpose, we recall some concepts and terminology taken from the vast literature on rotation distance.

For a tree S and a vertex x, the two paths of vertices starting at the left child of x and following right pointers, or starting at the right child of x and following left pointers, are respectively called the left and right *forearms* of x. The lengths (number of vertices) of the left and right forearms are denoted by $\lambda_S(x)$ and $\rho_S(x)$, respectively. We also let $c_S(x) = \lambda_S(x) + \rho_S(x)$. It can be easily seen that with a left (respectively right) rotation rot(y,x) the value of $\lambda_S(x)$ (respectively $\rho_S(x)$) increases by one. For the tree S in Fig. 1, the forearms of vertex 5 are composed of vertices {3, 4} and 6, respectively. We have $\lambda_S(5) = 2$, $\rho_S(5) = 1$, $c_S(5) = 3$. After the rotation rot(5,3), the length of the right forearm of vertex 3 changes from 1 in S to 2 in T. When clear from context, we will omit the subscripts for these notations.

Following Ref. [5], it can be seen that for any pair of trees S,T and any vertex x we have $d(S,T) \leq 2n - c_S(x) - c_T(x) - 2$, where the underlying transformation starts by bringing x to the root of S, and then transforms the tree thus obtained into T. Except for limit cases that can be treated separately, it has been shown that there is always at least one vertex x with $c_S(x) + c_T(x) \geq 4$, hence the bound $d(S,T) \leq 2n - 6$ is obtained. Our algorithm will be inspired by this approach.

Another important fact is that for any vertex x of a tree S, the labels of the vertices of the subtree rooted at x form a closed interval $[l_x,r_x]$, occasionally denoted by INT$_S(x)$ or simply INT(x). After a single (left or right) rotation rot(x,y),

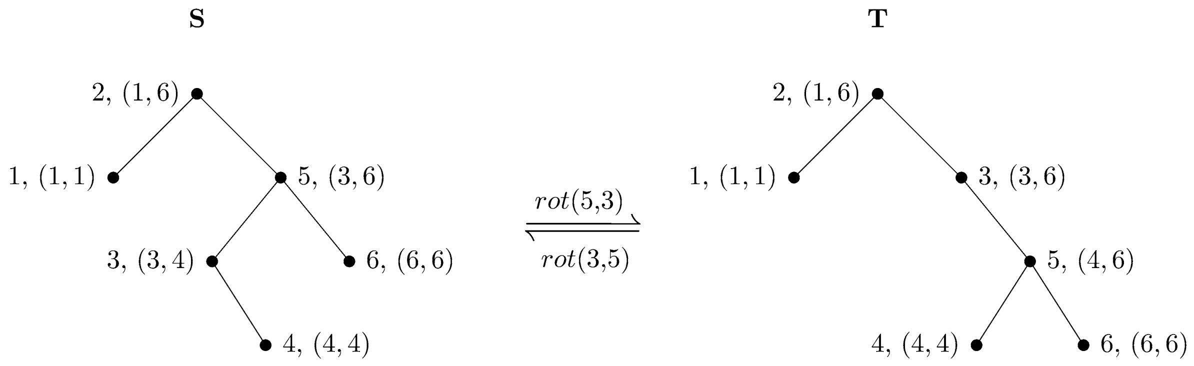

FIGURE 1. A right rotation rot(5,3) applied to the tree S produces the tree T. A left rotation rot(3,5) on T produces S again. The vertex numbering and the intervals associated to each vertex are also shown.

only one extreme of INT(x) and one extreme of INT(y) change, while all the other intervals in the tree remain unchanged (see Fig. 1). The following consequence of this will prove useful:

PROPERTY 1. *[16] In a right rotation rot(x,y) (y $<$ x), the values of l_x and r_y increase while the values of r_x and l_y remain unaffected. In a left rotation rot(x,y) (y $>$ x), the values of r_x and l_y decrease while l_x and r_y remain unaffected.*

3. AN UPPER BOUND FOR THE ROTATION DISTANCE

Our goal is finding a transformation algorithm for two trees S and T of n vertices, where T is an almost complete binary tree, using as little computation time and memory as possible. This has applications in the maintenance and manipulation of large data structures and files. For this reason, our algorithms are based on rotations. For $n \geq 11$, we are committed to improve the best known upper bound of $2n - \lceil \log_2 n \rceil$ rotations that can be derived from Ref. [14]. For $n < 11$ the problem can be easily solved, see Refs [2, 5].

For all trees, we know that $d(S,T) \leq 2n - 6$. One may expect that, if T is almost complete, the problem is somehow easier. However, finding an optimal strategy remains challenging even for this case. The basic structure of our transformation is given in Algorithm A_1 of Fig. 2 expressed in C-style pseudo-code. We assume that n is known.

The tree T built by A_1 has all the vertices at level h in the leftmost positions. The algorithm can be improved by relaxing this condition, as will be discussed later. Recalling that $2^{h-1} \leq n \leq 2^h - 1$, it can be easily seen that the root of T has label root$_T$ given by

$$\text{root}_T = \begin{cases} n - 2^{h-2} + 1 & \text{for } 2^{h-1} \leq n \leq 2^{h-1} + 2^{h-2} - 2 \\ 2^{h-1} & \text{for } 2^{h-1} + 2^{h-2} - 1 \leq n \leq 2^h - 1 \end{cases} \qquad (1)$$

For example for $n = 10$, hence $h = 4$, we have root$_T = 10 - 2^2 + 1 = 7$, with three leaves 1, 3 and 5 in the leftmost positions of level 4. For $11 \leq n \leq 15$, i.e. when all the leaves in the left half of T have been filled up, we have root$_T = 2^3 = 8$.

The current tree under transformation in algorithm A_1 is denoted by Q. The proper root for T is computed and located in Q, and is then lifted to the right position (step 4). Then, all the vertices in the left (resp. right) subtree of this root are brought into the left (resp. right) forearm of the root (step 5). Finally, the vertices in these forearms that must be in other positions in T are put in place by the two algorithms A_L, A_R of Fig. 3 (step 6).

To understand the functioning of the algorithm, we must specify how the operations **move** (steps 4 and 5) and **remove** (step 6) are done by rotations. The two **move** operations bring all the vertices of Q (except for its root) in the forearms of the root. In general, k vertices are moved into a chain C applying a rotation to each vertex v adjacent to C. If v has children,

```
algorithm $\mathcal{A}_1(S, n, T)$
1. denote $S$ as $Q$;
2. compute $root_T$ as in equation (1);
3. find $root_T$ in $Q$;
4. if ($root_T$ is at level $k > 1$)
        move $root_T$ to the root of $Q$ with $k - 1$ rotations, to get tree $Q$;
5. let $\mathcal{L}_Q, \mathcal{R}_Q$ be the left and right forearms of $root_T$ in $Q$,
      $X_Q$ be the set of vertices $v \in Q$, $v < root_T$, $v \notin \mathcal{L}_Q$,
      $Y_Q$ be the set of vertices $w \in Q$, $w > root_T$, $w \notin \mathcal{R}_Q$;
        move the vertices of $X_Q$ into $\mathcal{L}_Q$ with $|X_Q|$ rotations;
        move the vertices of $Y_Q$ into $\mathcal{R}_Q$ with $|Y_Q|$ rotations;
6. let $\mathcal{L}_T, \mathcal{R}_T$ be the left and right forearms of $root_T$ in $T$,
      $X_T$ be the set of vertices $v \in T$, $v < root_T$, $v \notin \mathcal{L}_T$,
      $Y_T$ be the set of vertices $w \in T$, $w > root_T$, $w \notin \mathcal{R}_T$;
        remove the vertices of $X_T$ from $\mathcal{L}_Q$ via algorithm $\mathcal{A}_L(Q)$;
        remove the vertices of $Y_T$ from $\mathcal{R}_Q$ via algorithm $\mathcal{A}_R(Q)$;
// $Q$ is equal to $T$ after step 6.
```

FIGURE 2. Basic structure of the transformation of an arbitrary binary tree S into an almost complete binary tree T. Algorithms A_L and A_R are shown in Fig. 3.

```
algorithm $\mathcal{A}_L(Q)$
1. $s = h$;
2. if $(2^{h-1} \le n \le 2^{h-1} + 2^{h-2} - 2)$
        $k = n - 2^{h-1} + 1$;
        for $(i = 1,\ i \le k,\ i{+}{+})$ $rot(\mathcal{L}_Q(i), \mathcal{L}_Q(i+1))$;
        $s = h - 1$;
// now $|\mathcal{L}_Q| = 2^{s-1} - 1$ and the chain must be transformed into
    a complete binary tree;
3. for $(j = s - 1,\ j \ge 1,\ j - -)$
        $k = 2^j - 1$;
        for $(i = 1,\ i \le k,\ i{+}{+})$ $rot(\mathcal{L}_Q(i), \mathcal{L}_Q(i+1))$;

algorithm $\mathcal{A}_R(Q)$
1. $s = h$;
2. if $(2^{h-1} + 2^{h-2} \le n \le 2^h - 2)$
        $k = n - 2^{h-1} - 2^{h-2}$;
        $p = n - 2^{h-1}$;     //$p$ is the length of $\mathcal{R}_Q$
        for $(i = p - 2,\ i \ge p - 2k,\ i = i - 2)$ $rot(\mathcal{R}_Q(i), \mathcal{R}_Q(i+1)))$;
        $s = h - 1$;
// now $|\mathcal{R}_Q| = 2^{s-1}$ and the chain from $\mathcal{R}_Q(1)$ to $\mathcal{R}_Q(2^{s-1} - 1)$ must
    be transformed into a complete binary tree;
3. for $(j = s - 1,\ j \ge 1,\ j - -)$
        $k = 2^j - 1$;
        for $(i = 1,\ i \le k,\ i{+}{+})$ $rot(\mathcal{R}_Q(i), \mathcal{R}_Q(i+1))$;
```

FIGURE 3. Constructing an almost complete binary tree T with $2^{h-1} \le n \le 2^h - 1$ from the tree Q built with steps 1–5 of algorithm A_1, using the same notation. $L_Q(i)$ and $R_Q(i)$ denote the ith vertices of the two chains.

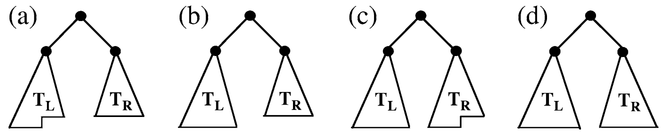

FIGURE 4. The four cases for T in algorithms A_L and A_R corresponding to: (**a**) $2^{h-1} \le n \le 2^{h-1} + 2^{h-2} - 2$; (**b**) $n = 2^{h-1} + 2^{h-2} - 1$; (**c**) $2^{h-1} + 2^{h-2} \le n \le 2^h - 2$; (**d**) $n = 2^h - 1$. The subtrees of the root are complete except for cases (a) and (c) where, respectively, the subtrees T_L and T_R are almost complete.

after the rotation they become in turn adjacent to C, so a total of k rotations are needed. Step 5 then requires $|X_Q| + |Y_Q|$ rotations. The two **remove** operations reconstruct the left and right subtrees of the root of T by algorithms A_L and A_R, as will now be explained.

We consider the four cases illustrated in Fig. 4. Algorithm A_L constructs the left subtree T_L of the root from the left forearm L_Q of Q, and distinguishes between case (a), where step 2 of the algorithm applies, and cases (b)–(d) where step 2 is skipped. In fact in step 2, the vertices of L_Q that will be leaves in the last level of T_L are detached from the chain. Then, as for cases (b)–(d), the complete subtree T_L is built from the vertices of L_Q in step 3. Let L_T be the forearm of the root of T. At each iteration of the **for** cycle for j, $2^j - 1$ vertices of $T_L - L_T$ to be assigned to levels h, $h - 1$,..., (respectively, to levels $h - 1$, $h - 2$,... in case b) are removed from L_Q by rotations.

Algorithm A_R is quite similar. It distinguishes between cases (c) and (a), (b), (d) and operates on the right forearm R_Q of Q examining the chain bottom-up, to build T_R. An upper bound for $d(S,T)$ is then given in the following:

Theorem 1. *Algorithm A_1 transforms an arbitrary binary tree S of n vertices, $2^{h-1} \le n \le 2^h - 1$, into an almost complete binary tree T of height h, in $\Theta(n)$ time and $\Theta(1)$ space, requiring exactly $2n - 2\lfloor \log_2 n \rfloor - c_S(\mathrm{root}_T) + p$ rotations, where*

$$p = \begin{cases} +1, & \text{if } 2^{h-1} \le n \le 2^{h-1} + 2^{h-2} - 2 \\ 0, & \text{if } n = 2^{h-1} + 2^{h-2} - 1 \\ -1, & \text{if } 2^{h-1} + 2^{h-2} \le n \le 2^h - 1. \end{cases}$$

Proof. Correctness: (A_1 builds an almost complete tree T). The basic structure of Fig. 2 is straightforward. The correctness of the **move** operations of step 5 has been already discussed, showing how a whole tree can be transformed via rotations into a right or left chain. With some attention, we can observe that the **remove** operations done by A_L and A_R are the same as the **move** operations of step 5 but performed in reverse order.

Complexity: Steps 2 and 3, i.e. finding root_T in S, are done with straightforward computation, with an in-order traversal of S. Let r the number of rotations of step 4 to bring root_T at the root of Q. Now $c_Q(\mathrm{root}_T) = c_S(\mathrm{root}_T) + r$, hence $m_1 = n - c_S(\mathrm{root}_T) - r - 1$ is the number of vertices of Q not in the root or in L_Q and R_Q (see Ref. [5]). Step 5 performs m_1 rotations to bring all these vertices into the forearms L_Q, R_Q of Q, hence the overall number of rotations performed by the end of step 5 is $m_2 = r + m_1 = n - c_S(\mathrm{root}_T) - 1$. Tree Q now consists of the root and its forearms.

Step 6 removes from L_Q and R_Q, the vertices not belonging to L_T and R_T, making one rotation for each vertex. The total number of such vertices is $m_3 = n - \lfloor \log_2 n \rfloor + p$, where p takes the values stated above. This can be verified in Fig. 4, where $p = +1$ in case (a), $p = 0$ in case (b) and $p = -1$ in cases (c) and (d). The total number of rotations is then $m_2 + m_3$ and the theorem follows. □

4. REFINING THE UPPER BOUND

Starting from the early paper [2], it is known that if S and T contain e *equivalent* subtrees (i.e. subtrees containing the same subsets of vertices), the transformation of S to T can be obtained through e independent pairwise transformations of the equivalent subtrees. This has an immediate impact on lowering the upper bound for $d(S,T)$ (e.g. see Ref. [8]). Since two equivalent subtrees S_1 and T_1 may contain other equivalent subtrees S_2 and T_2, the portions $S_1 - S_2$ and $T_1 - T_2$ are in fact trees but not whole subtrees of S and T. The standard transformation algorithm is then applied independently to such portions. To improve the upper bound for rotation distance, we consider only the equivalent subtrees that are identical and maximal, i.e. not contained in larger identical subtrees. This allows using an extension of algorithm A_1.

Given S and T, all the pairs of maximal identical subtrees can be found in linear time exploiting the vertex intervals (see Section 2). In fact for any vertex v of T, $\mathrm{INT}_T(v)$ is straightforwardly determined by the value of n, while $\mathrm{INT}_S(v)$ can be found with a post-order traversal of S. Identical subtrees, if any, are detected in the same traversal. A subtree of S rooted at v is identical to the subtree of T also rooted at v if $\mathrm{INT}_S(v) = \mathrm{INT}_T(v)$ and the left and right subtrees of v in S have been found identical to the corresponding subtrees of v in T. If identical, the subtrees rooted at v are (temporarily) declared maximal and their left and right subtrees lose this property. After $\mathrm{INT}_S(v)$ has been built, the intervals of its children are no longer needed, then the whole process can be executed in constant space plus n one-bit flags to mark the roots of the maximal identical subtrees in S.

Let $f \geq 0$ be the number of pairs of maximal identical subtrees. For $f = 0$, algorithm A_1 applies as it is and the upper bound on $d(S,T)$ of Theorem 1 holds unchanged. For $f > 0$, the bound can be improved by an extension of algorithm A_1. To see how this can be done, a new observation is needed:

Observation 1. *Let S_1 and T_1 be two maximal identical subtrees of S and T rooted at u, and let π be the path in S from root_T to the root of S. Then S_1 does not share a vertex with π unless $u = \mathrm{root}_T$ (in which case $S = T$ and π reduces to u).*

In fact for $r \neq \mathrm{root}_T$, we have $\mathrm{root}_T \notin T_1$. By contradiction, for a vertex v of S_1 let $v \in \pi$. Then, v must be an ancestor of root_T in S thus implying $\mathrm{root}_T \in S_1$, against the hypothesis that $S_1 = T_1$.

For $f > 0$, the new algorithm A_2 is an extension of A_1, as follows. Let S_1,T_1; S_2,T$_2$… S_f,T_f be the pairs of maximal identical subtrees. If S,T is one such a pair, i.e. $S = T$, no operation is done (note that in this case algorithm A_1 would be fully executed, dismounting S and remounting it as it was). If $S \neq T$, due to Observation 1 the same operations of algorithm A_1 can be executed up to step 4 without any interference between the rise of root_T up to the root of Q and the subtrees $S_1,S_2,\ldots,S_f$ in Q. Then, step 5 is done without transferring the vertices of S_1, $S_2,\ldots,S_f$ into L_Q and R_Q since such vertices are already in the correct positions of T. And then step 6 is done, removing from L_Q and R_Q the vertices that do not belong to L_T and R_T. Letting $n_i = |S_i|$, $1 \leq i \leq f$, we have:

Theorem 2. *For $f > 0$ and $S \neq T$, algorithm A_2 transforms S into T in $\Theta(n)$ time and* n *bits of space, requiring $2n - 2\lfloor \log_2 n \rfloor - c_S(\mathrm{root}_T) - 2\sum_{i=1}^{f} n_i + p$ rotations where $|p| \leq 1$.*

Proof. Correctness: Similar to the one of Theorem 1 with an additional observation. Consider the maximal identical subtrees as collapsed into single leaves. Algorithm A_1 would insert these leaves into L_Q and R_Q and then remove them in the first round of removal, leaving them attached to the proper parents in T. Algorithm A_2 would instead skip the move and remove operations for these leaves without changing their parents. Hence, the maximal identical subtrees are also properly positioned in the final tree.

Complexity: As in Theorem 1, noting that all the vertices in S_1, $S_2,\ldots$, S_f are not moved into and removed from L_Q and R_Q, then decreasing the number of rotations by $\sum_{i=1}^{f} n_i$. □

If pairs of maximal identical subtrees of large size are present, the transformation process is considerably faster than that of algorithm A_1. As this may not occur frequently, a different refinement of the method is also considered, which works on maximal equivalent subtrees, which are pairs of subtrees of S and T that are maximal and contain the same vertices (hence their roots have equal intervals) but are not necessarily identical. For all vertices $v \in T$ the intervals $\text{INT}_T(v)$ and for all vertices $u \in S$ the intervals $\text{INT}_S(u)$ are generated as explained before. Given $\text{INT}_S(u)$, deciding if there is a vertex v such that $\text{INT}_S(u) = \text{INT}_T(v)$ can be done in constant time. Then, the g pairs S_1,T_1; S_2,T_2;…; S_g,T_g of the maximal equivalent subtrees, if any, are built in $\Theta(n)$ time.

The extension of algorithm A_2, denoted A_3, is as follows: For each pair S_i,T_i with $S_i \neq T_i$, transform S_i into T_i with algorithm A_1 so S_i becomes identical to T_i, then apply A_2 as before. Letting $n_i = |S_i|$, $1 \le i \le g$, with straightforward calculations we have:

THEOREM 3. *Algorithm A_3 transforms S into T in $\Theta(n)$ time and n bits of space, requiring at most* $2n - 2\lfloor\log_2 n\rfloor - c_S(\text{root}_T) - 2\sum_{i=1}^{g}\lfloor\log_2 n_i\rfloor + g + 1$ *rotations.*

The advantage of algorithm A_3 over A_1 is given by the new term $-2\sum_{i=1}^{g}\lfloor\log_2 n_i\rfloor$ in the number of rotations. This term is clearly smaller than the term $-2\sum_{i=1}^{g} n_i$ for A_2, as expected, as extra work on the subtrees S_1, S_2… S_g is needed to make them identical to T_1, T_2… T_g. However, algorithm A_3 will apply more frequently because equivalent subtrees occur more frequently than identical ones.

5. LOWER BOUNDS

Before examining some particular cases of tree rotation distances, we first discuss existing results that will be useful in establishing some new bounds in Section 6.

Some parameters proved to be useful for studying lower bounds on the rotation distance (see Refs. [16, 17] for results on specific families of trees) clearly also apply to our case. In particular for two trees S and T of n vertices, and for any vertex x in S and T, $1 \le x \le n$, consider the intervals $\text{INT}_S(x)$ and $\text{INT}_T(x)$. Let L_+ (respectively L_-) be the number of vertices x such that l_x in $\text{INT}_S(x)$ is greater (respectively smaller) than l_x in $\text{INT}_T(x)$; and let R_+ (respectively R_-) be the number of vertices x such that r_x in $\text{INT}_S(x)$ is greater (respectively smaller) than r_x in $\text{INT}_T(x)$. We have from [16]:

$$d(S,T) \ge \max(L_+, R_+) + \max(L_-, R_-) \tag{2}$$

Furthermore, if S and T have e equivalent subtrees we have from [8]:

$$d(S,T) \ge n - e - 1 \tag{3}$$

In any tree, vertices x and $x + 1$ are called *consecutive*. Note that two consecutive vertices must lie in the same path from the root to a leaf, both in S and in T. If traversing S from top to bottom x is encountered before (respectively after) $x + 1$, and if traversing T from top to bottom x is encountered after (respectively before) $x + 1$, the pair x, $x + 1$ is called an *inversion*. Letting $I(S,T)$ be the number of inversions between S and T, $0 \le I(S,T) \le n - 1$, we have from [16]:

$$d(S,T) \ge I(S, T) \tag{4}$$

because a direct rotation between any pair of vertices forming an inversion must be executed.

No specific lower bound for $d(S,T)$, where T is an almost complete binary tree, has been found thus far. In two particular cases, however, we can do better, as shown in the next section.

6. TWO SIGNIFICANT CASES

The lower bounds mentioned in the previous section do not match the upper bounds of Theorems 1, 2 and 3. However, for particular families of trees we may be able to prove tighter bounds, and in some case even matching bounds. We consider two cases where T is a completely balanced tree of $n = 2^h - 1$ vertices, and S is a *zig-zag tree* as shown in Fig. 5, or S is a *quasi-complete binary tree* as shown in Fig. 6, whose structure is quite similar to the one of T except that vertex $2^{h-1} - 1$ instead of 2^{h-1} is at the root. For both, we show that a large number of rotations are needed for the transformation, and more importantly, we are able to determine a very close lower bound for the first case and a matching lower bound for the second.

Note that none of the other known bounds of Refs. [6–11] give better results than ours for these two cases. It is worth noting that the zig-zag structure not necessarily extended to a whole tree has been originally studied by Lucas [18] for a transformation between two trees with only one leaf.

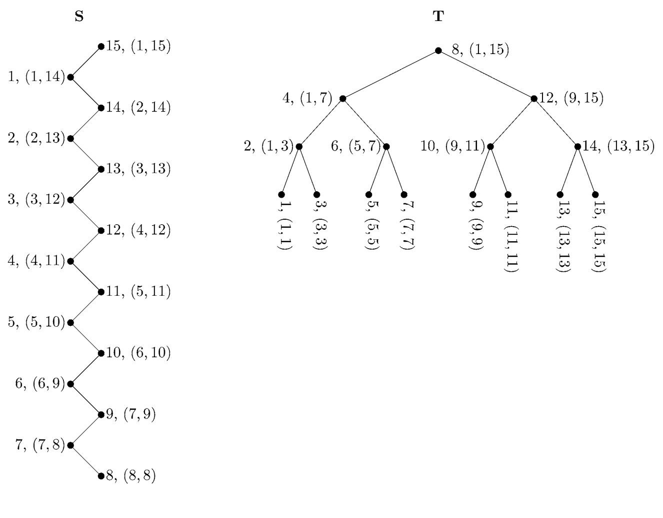

FIGURE 5. For $n = 15$, S is a zig-zag tree, T is a completely balanced tree of $n = 2^4 - 1 = 15$. The intervals associated to their vertices are also indicated.

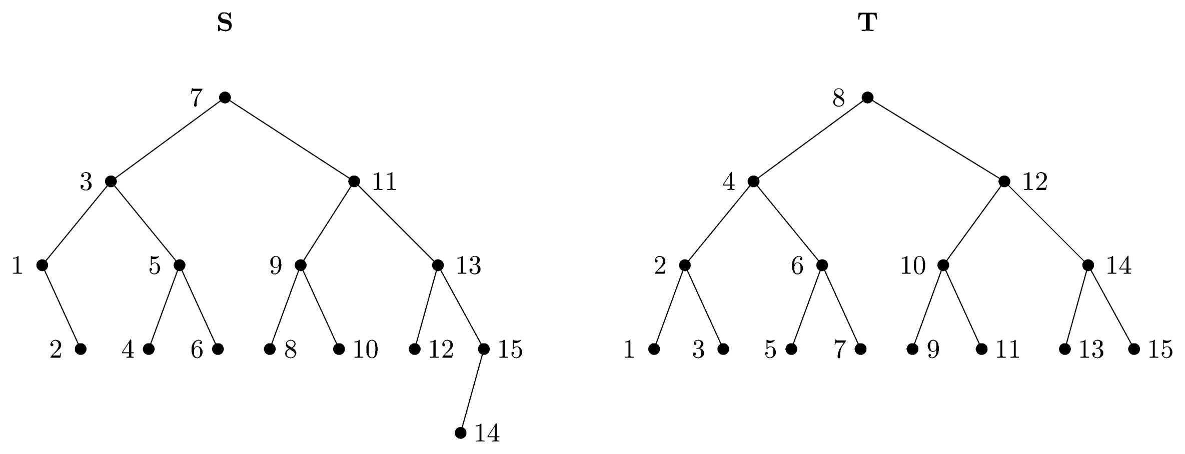

FIGURE 6. A quasi-complete binary tree S and a complete binary tree T with $n=2^h - 1$ vertices (in the example $h=4$). All pairs of consecutive vertices form an inversion, then the number $I(S, T)$ has maximum value $n - 1$.

6.1. Zig-zag and complete trees

Let S be a zig-zag tree and T be a completely balanced tree, with $n = 2^h - 1$ vertices. Without loss of generality, we consider the case where the root of S contains the maximum element (similar reasonings hold for the case where the root corresponds to the minimum). In this case, it is easy to check that $L_+ = 2^{h-2}$, $L_- = 2^{h-1} - 1$, $R_+ = 2^{h-1} - 1$, $R_- = 2^{h-2}$. For example, for the two trees of Figure 5, we have that the set of vertices {2, 4, 6, 8} contributes to L_+, hence $L_+ = 4$. The set of vertices {9, 10, 11, 12, 13, 14, 15} contributes to $L_- = 7$. Similarly, {1, 2, 3, 4, 5, 6, 7} contributes to $R_+ = 7$ and {8, 10, 12, 14} contributes to $R_- = 4$. Applying the lower bound of relation (2), we obtain

$$d(S,T) \geq (2^{h-1} - 1 + 2^{h-1} - 1) = 2^h - 2 = n - 1.$$

As far as the other criteria mentioned in Section 5, it is easy to check that there are no equivalent edges hence the lower bound of relation (3) also reduces to $n - 1$; and there are $(n + 1)/2$ inversions (due to vertices {9, 10, 11, 12, 13, 14, 15} in the example) so the bound of relation (4) is even less significant. However, another argument can be applied that, to our knowledge, it has never been considered in the literature.

To establish a lower bound on $d(S,T)$, let h_S and h_T be lengths of the longest paths root to leaf in S and T respectively, and assume $h_S > h_T$. Choose a leaf e in S such that the path σ from the root to e has length greater than h_T. σ is called the *spine* of S; the length of σ is denoted by λ and its top and bottom vertices are called the *extremes*. Since σ may be changing along a transformation, it will be identified by its extremes, namely the current root of the tree, and the bottom extreme determined as follows. Initially, the bottom extreme is e. As far as no rotation rot(f,e) is done, the bottom extreme does not change. If a rotation rot(f,e) is done, e is raised one level above f and f becomes the bottom extreme. If f has a second child before the rotation, it is not a leaf after the rotation, hence the bottom extreme of σ is not necessarily a leaf. Note that the extremes of σ, and then the spine itself, are uniquely defined at any step. A zig-zag tree has an initial spine consisting of the whole tree, while in a completely balanced tree the initial spine can be any path root to leaf.

Our lower bound argument is based on the number of rotations needed to reduce λ to at least h_T. For this purpose, we need some further definitions. Consider three consecutive vertices x_{i-1}, x_i and x_{i+1} of the spine.

- x_i is a *flat* if x_i is the left child of x_{i-1} and x_{i+1} is the left child of x_i, or x_i is the right child of x_{i-1} and x_{i+1} is the right child of x_i. The number of flats of σ is denoted by β.
- x_i is a *cusp* if x_i is the left child of x_{i-1} and x_{i+1} is the right child of x_i, or x_i is the right child of x_{i-1} and x_{i+1} is the left child of x_i. The number of cusps of σ is denoted by γ.

Note that the extremes of σ are neither flats nor cusps. We have

$$\lambda = \beta + \gamma + 2.$$

In a zig-zag tree, all vertices except the root and the leaf are cusps, with $\lambda = n$, $\beta = 0$ and $\gamma = n - 2$. The effect of rotations on the values of λ and β and γ plays a basic role in the lower bound computation. By direct inspection, we have:

PROPERTY 2. *A rotation rot(u,v) with u, $v \notin \sigma$ does not change σ (then λ and β and γ do not change).*

PROPERTY 3. *A rotation rot(u,v) with $u \in \sigma$ and $v \notin \sigma$ inserts v in σ. We have*

(i) If u is a flat, v becomes a flat. If u is a cusp, v becomes a cusp and u becomes a flat. In both cases, λ and β increase by 1 and γ does not change. See Fig. 7.
(ii) If u is the root, then v becomes the root and u becomes a flat. λ and β increase by 1 and γ does not change.
(iii) If u is the bottom extreme, v is inserted above it and we have
   (1) if u is the left (respectively, right) child of a vertex w and v is the left (respectively, right) child of u, then v becomes a cusp. λ and γ increase by 1 and β does not change;
   (2) if u is the left (respectively, right) child of a vertex w and v is the right (respectively, left) child of u, then v becomes a flat. λ and β increase by 1 and γ does not change.

Property 4. *For a rotation rot(u,v) with u,v $\in \sigma$, we have*

(i) If v is a flat, u is extracted from σ. λ and β decrease by 1 and γ does not change. See Fig. 7.
(ii) If v is a cusp, the vertices of σ remain the same although in a different order and λ does not change. In particular, let w be the child of v in σ:
   (1) if u and w are flats then u, v and w become cusps. β decreases by 2 and γ increases by 2;
   (2) if u (respectively w) is a cusp and w (respectively u) is a flat, then u and w (respectively u and v) become cusps and v (respectively w) becomes a flat. β and γ do not change;
   (3) if u and w are both cusps, v remains a cusp and u and w become flats. β increases by 2 and γ decreases by 2. See Fig. 7.
(iii) If v is the bottom extreme, the vertices of σ remain the same although in a different order and λ does not change (in particular u becomes the bottom extreme). We have
   (1) if u is a cusp v becomes a flat, β increases by 1 and γ decreases by 1;
   (2) if u is a flat v becomes a cusp, β decreases by 1 and γ increases by 1.

Note that a rotation rot(u,v) with $u \notin \sigma$ and $v \in \sigma$ cannot occur because $v \in \sigma$ implies that also its parent is in σ.

To transform S into T, path σ must become no longer than the longest path root to leaf in T. From the above properties, we see that λ may decrease by at most 1 for each rotation (Property 4-(i)), and we immediately have

$$d(S, T) \geq \lambda - h_T. \qquad (5)$$

where λ is the initial length of the spine. Relation 5 is quite weak, for example the difference $\lambda - h_T$ has the maximal possible value $n - \log_2(n+1)$ for zig-zag versus completely balanced trees, and this result is worse than the bounds already found. However, referring to the parameters λ, β and γ of the initial spine we can prove a much stronger lemma that may be useful if the two trees have strongly different heights. That is $h_S \gg h_T$, as in zig-zag versus completely balanced trees. We have:

Lemma 6.1. (i) $d(S,T) \geq \lambda - h_T$, for $\beta \geq \lambda - h_T$,
(ii) $d(S, T) \geq \lambda - h_T + \lfloor(\lambda - h_T - \beta)/2\rfloor$, for $\beta < \lambda - h_T$.

Proof. (i) For $\beta \geq \lambda - h_T$, the immediate bound of relation 5 is confirmed. Note that this bound is tight because it equals the number of rotations of a transformation algorithm the extracts $\lambda - h_T$ vertices from the spine by doing rotations on any subset of $\lambda - h_T$ flats (Property 4-(i)).

(ii) For $\beta < \lambda - h_T$, β rotations on the flats are not sufficient and at least $\lambda - h_T - \beta$ new flats must be created by removing the same number of cusps. To this end at least $\lfloor(\lambda - h_T - \beta)/2\rfloor$ rotations on the cusps are needed, each of which generates two new flats (Property 4-(ii)-3). The only alternative would be a rotation rot(u,v) where v is the bottom extreme of σ and u is a cusp (Property 4-(iii)-1); however, this operation is weaker than the previous one since only one new flat is created. With any other rotation (see Properties 2, 3, 4), either the values of λ and β do not change, or these values change but the difference $\lambda - \beta$ does not decrease, while we must reach the bound $\lambda - \beta = h_T$. □

Theorem 4. *Given a balanced tree T and a zig-zag tree S on n = $2^h - 1$ vertices,* $d(S, T) \geq 3(n-1)/2 - 3\lfloor\log_2 n\rfloor/2 - 1/2$.

Proof. Apply case (ii) of Lemma 6 with the values: $\lambda = n$, $\beta = 0$ and $h_T = \log_2(n+1)$. We have $d(S, T) \geq$

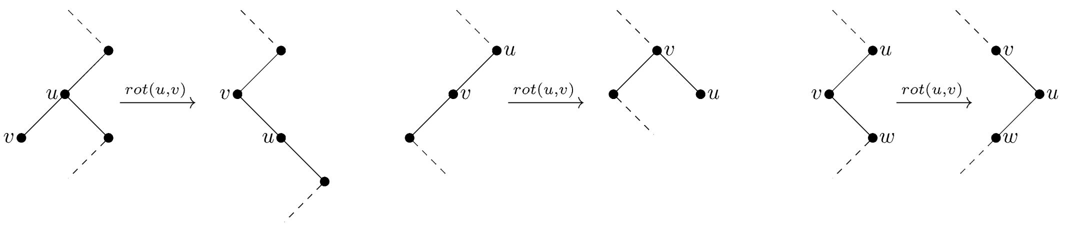

FIGURE 7. Examples of rotations to illustrate Property 3-(i) with u cusp, Property 4-(i) and Property 4-(ii)-3, respectively.

$n - \log_2(n+1) + \lfloor (n - \log_2(n+1))/2 \rfloor = 3(n-1)/2 - 3\lfloor \log_2 n \rfloor/2 - 1/2$, where the term $-1/2$ arises if $\lfloor \log_2 n \rfloor$ is odd. □

A simple, yet efficient, algorithm provides an upper bound very close to the lower bound of Theorem 4. Informally, the algorithm performs rotations on the zig-zag tree to build the levels of the balanced tree from the bottom up with a procedure similar to the one discussed in the lower bound proof, focusing now the attention on the leaves of the balanced tree that are paired, lowest with highest (numbered according to an in-order traversal), second lowest with second highest, etc. The rotations on the zig-zag tree produce these pairs as new leaves, and hence generate equivalent edges. Both trees can be pruned at these equivalent edges, reducing the remaining problem. Once an entire level (relative to the balanced tree) has been pruned, we have in effect a new instance with a smaller balanced tree and the corresponding zig-zag.

As already noted, there are two possible zig-zag trees with n vertices. Either the root contains the maximum element (see Fig. 5), or the root contains the minimum element, and the tree is the mirror image the first case. As done for the lower bound, we will assume that the zig-zag tree is in the first category. The rotations needed for the second category are similar but 'mirrored', the details will be given later.

Let $n = 2^h - 1$, T be a completely balanced tree with n vertices and S a zig-zag tree with n vertices. We pair the vertices of S according to their labelling in an in-order traversal where vertex i is paired with vertex $n - i + 1$. The corresponding algorithm A_4 is given in Fig. 8.

It is evident from the structure of the zig-zag tree that vertex i, where $i \leq \lfloor n/2 \rfloor$, has as a right child vertex $n - i$, which in turn has as a left child vertex $i + 1$. The parent of vertex i is vertex $n - i + 1$. Assuming we are not in the degenerate case where $n - i = i + 1$ (which occurs at the bottom of the zig-zag when $i = \lfloor n/2 \rfloor$ and hence $n - i + 1 = \lceil n/2 \rceil + 1$), then rotating vertices i and $n - i$ produces a left chain of length two and a right chain of length two, from which the further two rotations produce two leaves, precisely the vertices i and $n - i + 1$, which are leaves in T and hence generate equivalent edges that can be pruned. The action of these rotations is illustrated in Fig. 9.

Theorem 5. *Given a balanced tree T and a zig-zag tree S on $n = 2^h - 1$ vertices, $d(S, T) \leq 3(n-1)/2 - \lfloor \log_2 n \rfloor$.*

Proof. We proceed by induction on h.

For $h = 1$ (and hence $n = 1$), Algorithm A_4 performs no rotations and $3(n-1)/2 - \lfloor \log_2 n \rfloor = 0$.

Assume that the algorithm performs at most $3(2^h - 2)/2 - \lfloor \log_2(2^h - 1) \rfloor$ rotations for all $h \leq m$.

Let $h = m + 1$. T thus has 2^m leaves. Algorithm A_4 pairs these leaves and for each pair except one performs three rotations in S. For the remaining pair (the innermost pair), the algorithm performs two rotations in S. This gives a total of $3 \cdot 2^{m-1} - 1$ rotations after which all the leaves can be pruned, leaving T as a balanced tree with $2^m - 1$ vertices and S as the corresponding zig-zag tree. By the inductive assumption, we can solve this sub-problem using at most $3(2^m - 2)/2 - \lfloor \log_2(2^m - 1) \rfloor$ rotations, giving a total of

$$\begin{aligned} &3(2^m - 2)/2 - \lfloor \log_2(2^m - 1) \rfloor + 3 \cdot 2^{m-1} - 1 \\ &= 3(2^{m+1} - 2)/2 - \lfloor \log_2(2^{m+1} - 1) \rfloor \\ &= 3(n-1)/2 - \lfloor \log_2 n \rfloor. \end{aligned}$$

□

6.2. Quasi-complete and complete trees

Let S be a quasi-complete binary tree and T be a complete binary tree with $n = 2^h - 1$ vertices (an example is given in

```
algorithm $\mathcal{A}_4(S,T)$
1. while $T$ has more than one vertex
2.     for Each pair of vertices $i$ and $n-i+1$ where $i$ is the lowest numbered unmarked leaf in $T$
3.         if $i+1 = n-i$
               $rot_S(i, n-i)$;
               $rot_S(n-i+1, n-i)$;
           else
               $rot_S(i, n-i)$;
               $rot_S(i, i+1)$;
               $rot_S(n-i+1, n-i)$;
4.         mark vertices $i$ and $n-i+1$ in $S$ and $T$;
5.     prune marked vertices from $S$ and $T$;
```

FIGURE 8. Algorithm for converting a zig-zag tree S into an balanced binary tree T.

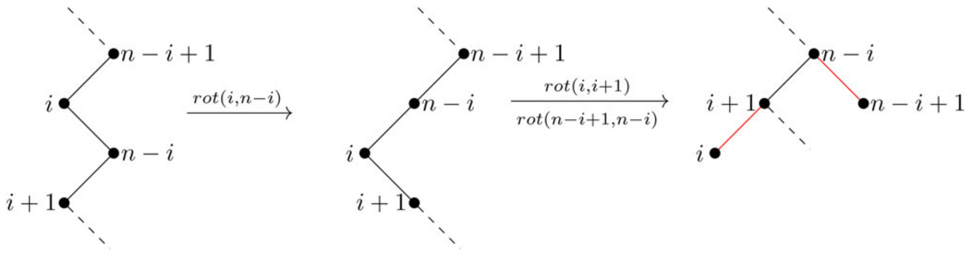

FIGURE 9. The basic operation of the algorithm. The first rotation creates two chains, from which the following two rotations (the order is in fact unimportant) produce leaves incident to equivalent edges. The resulting equivalent edges, which will be pruned by the algorithm, are the edges $(i+1,i)$ and $(n-i, n-i+1)$.

Fig. 6). Referring to Equation (2), it can be easily found that $L_+ = L_- = R_+ = R_- = 2^{h-1} - 1$, then $d(S, T) \geq n - 1$. Furthermore, it can be immediately checked that S and T do not contain equivalent subtrees, and as a consequence, the lower bound given by Equation (3) reduces to $n - 1$. There are also exactly $n - 1$ inversions, and even the lower bound given by Equation (4) coincides with the others. However, this is a starting point to apply a subtler argument to sharpen the bound.

In fact note that in order to flip, via rotation, two vertices $x, x+1$, which form an inversion they must be adjacent, so other rotations may be needed to bring them close. Some of these rotations may in turn be required to invert other pairs of consecutive vertices, but some may be needed only for bringing x adjacent to $x+1$. We use this requirement to sharpen the lower bound for quasi-complete trees, following a criterion given in Ref. [16]. We have:

Theorem 6. For a quasi-complete binary tree S and a complete binary tree T with $n = 2^h - 1$ vertices, we have $d(S, T) \geq 2^h + 2^{h-1} - 2h$.

Proof. Denoting by $I(h)$ and $E(h)$, respectively, the number of inversions and the minimum number of additional rotations to bring close all pairs vertices forming an inversion, we have $d(S, T) \geq I(h) + E(h)$, where $I(h) = 2^h - 2$. For vertices $x, x+1$ forming an inversion, let $\pi(x, x+1)$ be the path in S between them, and $\delta(x, x+1)$ be their distance. To compute $E(h)$, we must consider non-overlapping paths with $\delta > 1$: in case of overlapping we take the one with maximal length. In any such path, the number of rotations contributing to the value of $E(h)$ is given by δ less the number of pairs forming an inversion, including $x, x+1$ (i.e. the extremes of π). For the tree S of Fig. 5 with $h = 4$, path $\pi(6,7)$ adds 1 to the value of $E(4)$ since $\delta(6,7) = 3$ and the path contains the two inversions 5,6 and 6,7. Similarly path $\pi(7,8)$ adds 1 to $E(4)$, while all the other paths do not contribute to $E(4)$ since either they have length 1, or contain a number of inversions equal to their length (e.g. see path $\pi(2,3)$). We must evaluate $E(h)$ for any value h.

By direct inspection, it can be immediately verified that $d(S, T) = 2$ for $h = 2$, and $d(S,T) = 6$ for $h = 3$. That is, for $h = 2,3$ we have $E(h) = 0$ and $d(S,T) = I(h)$. For $h \geq 4$ the value of $E(h)$ is established by recurrence. The base for $h = 4$ is the one discussed in Fig. 5 where $E(4) = 2$. For $h>4$, note that the number of extra rotations needed in the two subtrees of the root is the same as for $h - 1$, while the new root $2^{h-1} - 1$ generates two new paths $\pi(2^{h-1} - 2, 2^{h-1} - 1)$ and $\pi(2^{h-1} - 1, 2^{h-1})$ of length $h - 1$, that add $h - 3$ each to the value of $E(h)$ (as the paths $\pi(6,7)$ and $\pi(7,8)$ do for $h = 4$). We then have $E(4) = 2$, $E(h) = 2E(h-1) + 2(h-3)$ for $h > 4$, which has closed form $E(h) = 2^{h-1} - 2(h-1)$ (this figure holds also for $h = 2,3$). As a conclusion, we have

$$\begin{aligned} d(S, T) \geq I(h) + E(h) &= 2^h - 2 + 2^{h-1} - 2(h-1) \\ &= 2^h + 2^{h-1} - 2h. \end{aligned}$$ □

A simple recursive algorithm A_5, given in Fig. 10, provides a tight upper bound. We describe the key details, sufficient to evaluate exactly the required number of rotations. A_5 is based on the transformation via rotation of two trees $\Gamma)(h)$, $\Delta(h$ of $2^h - 2$ vertices, one into the other. These trees have a particular shape. Both are complete binary trees with a missing leaf. In $\Gamma)(h$ (respectively $\Delta)(h)$, the leftmost (respectively rightmost) leaf is missing. For example, if vertex 15 is detached from the tree T of Fig. 5 we are left with $\Delta(4)$. Similarly if rot(15,14) is executed in S and then vertex 15 is detached we are left with $\Gamma(4)$.

From the operations above, we immediately write the recurrence

$$R(h) = \begin{cases} 1 & \text{if } h = 2 \\ 2R(h-1) + 2h - 3 & \text{for } h > 2 \end{cases}$$

which has closed form $R(h) = 2^h + 2^{h-1} - 2h - 1$. Summing the rotation in step 1 of Algorithm A_5, we have:

Theorem 7. *Algorithm A_5 transforms a quasi-complete binary tree S of $n = 2^h - 1$ vertices into a complete binary tree with $2^h + 2^{h-1} - 2h$ rotations.*

```
algorithm $\mathcal{A}_5(S, T)$
1. Perform the rotation $rot(2^h - 1, 2^h - 2)$;
    // In Figure 5, $rot(15, 14)$ would lift vertex 14 at the 4-th level and vertex 15
       would become its right child;
    // From now on vertex $2^h - 1$ (15 in the figure) is left attached to its parent and
       is disregarded in the following operations;
2. Transform the tree $\Gamma(h)$ consisting of $S$ deprived of vertex $2^h - 1$ into the
   corresponding $\Delta(h)$;
    // In Figure 5, $\Delta(4)$ would be $T$ deprived of vertex 15;
    // The transformation of $\Gamma(h)$ into $\Delta(h)$ is done recursively;
    Let $R(h)$ be the number of rotations needed:
    // For $h = 2$, that is the two trees contain vertices 1 and 2 only, one rotation
       suffices hence $R(2) = 1$;
    if $(h = 2)$ $R(2) = 1$
    else // $h > 2$ the following operations are done on $\Gamma(h)$:
        2.1 Lift the vertex $2^{h-1}$ to the root by $h - 1$ rotations
        the former root $2^{h-1} - 1$ becomes its left child;
        // In Figure 5 vertex 8 would be lifted at the root with
        // three rotations and vertex 7 would become its left child;
        // now vertex $2^{h-1} - 1$ has no right child. The left subtree of vertex $2^{h-1} - 1$
           and the right subtree of vertex $2^{h-1}$ have the form $\Gamma(h-1)$: call them $\Gamma_1$ and $\Gamma_2$;
        2.2 transform recursively $\Gamma_1(h-1)$ and $\Gamma_2(h-1)$ into the corresponding $\Delta_1(h-1)$
        and $\Delta_2(h-1)$;
        2.3 move down vertex $2^{h-1} - 1$ to become the rightmost leaf of its left subtree
        $\Delta_1(h-1)$, with $h - 2$ rotations.
```

FIGURE 10. Algorithm for the transformation via rotation of two trees $\Gamma(h)$, $\Delta(h)$ of $2^h - 2$ vertices, one into the other.

Theorems 6 and 7 show that the number of rotations needed is approximately equal to $3n/2 - 2\log_2 n$. This number, which depends on the nature of the rotations, may actually appear to be exceedingly large considering that S is somehow very close to a completely balanced tree. We will return on this point in the conclusions.

Note that both of the considered cases require $(3/2)n - O(\log_2 n)$ rotations and represent the worst case we have encountered in our study.

7. CONCLUDING REMARKS

We have studied the rotation distance $d(S, T)$ between an arbitrary binary tree S and a binary tree T completely or almost completely balanced, discussing three transformation algorithms of increasing sophistication. We also studied the two cases where S is a zig-zag tree or almost balanced binary tree and T is completely balanced. In the first case, we could derive very close lower and upper bounds for $d(S, T)$. In the second case, the exact rotation distance has been obtained and it turns out that this distance is surprisingly large.

Some remarks are still in order on how the proposed approach could be improved.

We have already remarked that the problem of transforming an arbitrary binary tree into a complete tree has raised attention in the field of data structure for the construction of efficient indexes. Transforming by rotation has the advantage of not requiring extra memory; however, minimizing the number of rotations is important. From the literature on rotation distance, however, we may not expect to easily solve this problem unless new strong results are found in that field. So we have merely proposed the best algorithms that we could find, in particular, exploiting features of the given tree S and of the target tree T.

Our algorithms could be improved working in three directions. First algorithms A_2 and A_3 make use of identical or equivalent subtrees in S and T that are detected and treated separately. Since such subtrees may in turn contain subtrees of the same type, the two algorithms could be reformulated to work recursively at the expense of some substantial complications, with the goal of further reducing the number of rotations.

Second, a preprocessing of tree S could generate identical or equivalent subtrees not present in the original form, with a number of rotations smaller than the one required by A_2 or A_3 applied on the new form of S. For example, this is the case when one rotation in S transforms a vertex x into a leaf that is also a leaf of T (hence the two subtrees consisting of x only become equivalent) then saving two successive rotations when applying A_2.

Third, for simplicity we have assumed that T is almost complete, meaning that the leaves at level h are placed in the leftmost positions of that level. For most applications, however, such leaves could be allocated in any positions of level h. So different forms of the target tree T could require less rotations, although deciding which is preferable requires substantial additional computation. With this in mind we have presented the two cases of Section 6 where a tree S that would be probably acceptable in many applications provably requires a large number of rotations to become perfectly balanced.

ACKNOWLEDGEMENT

The authors are very grateful to the referees for many useful comments and suggestions.

FUNDING

The research of Bernard Mans was partially supported by Australian Research Council (Grant DP130100237).

REFERENCES

[1] Culik, K. and Wood, D. (1982) A note on some tree similarity measures. *Inform. Process. Lett.*, **15**, 39–42.

[2] Sleator, D.D., Tarjan, R.E. and Thurston, W.R. (1988) Rotation distance, triangulations, and hyperbolic geometry. *J. Am. Math. Soc.*, **1**, 647–681.

[3] Pournin, L. (2014) The diameter of associahedra. *Adv. Math.*, **259**, 13–42.

[4] Mäkinen, E. (1988) On the rotation distance of binary trees. *Inform. Process. Lett.*, **26**, 271–272.

[5] Luccio, F. and Pagli, L. (1989) On the upper bound on the rotation distance of binary trees. *Inform. Process. Lett.*, **31**, 57–60.

[6] Baril, J.-L. and Pallo, J.-M. (2006) Efficient lower and upper bounds of the diagonal-flip distance between triangulations. *Inform. Process. Lett.*, **100**, 131–136.

[7] Chen, Y., Chang, J., and Wang, Y. (2005) An efficient algorithm for estimating rotation distance between two binary trees. *Int. J. Comp. Math.*, **82**, 1095–1106.

[8] Cleary, S., and St. John, K.(2010) A linear-time approximation algorithm for rotation distance. *J. Graph Algorithms Appl.*, **14**, 385–390.

[9] Lucas, J. (2010) An improved kernel size for rotation distance in binary trees. *Inform. Process. Lett.*, **110**, 481–484.

[10] Pallo, J. (2000) An efficient upper bound of the rotation distance of binary trees. *Inform. Process. Lett.*, **73**, 87–92.

[11] Rogers, R. (1999) On finding shortest paths in the rotation graph of binary trees. *Congr. Numer.*, **197**, 77–95.

[12] Andersson, A. (1989) Improving partial rebuilding by using simple balance criteria. *Workshop on Algorithms and Data Structures (WADS) '89*, Ottawa, Canada, 17–19 August, pp. 393–402. Springer, New York.

[13] Galperin, I. and Rivest, R.L. (1993) Scapegoat trees. *Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, Austin, Texas, 25–27 January, pp. 165–174. SIAM, Philadelphia, PA.

[14] Andersson, A. (2004) Balanced binary search trees. In Mehta, D. P. and Sahni, S. (eds), *Handbook of Data Structures and Applications*, chapter 10. CRC Press, Cleveland, OH.

[15] Stout, Q.F. and Warren, B. L. (1986) Tree rebalancing in optimal time and space. *Commun. ACM*, **29**, 902–908.

[16] Luccio, F., Mesa Enriquez, A. and Pagli, L. (2010) Lower bounds for the rotation distance of binary trees. *Inform. Process. Lett.*, **110**, 934–938.

[17] Dehornoy, P. (2010) On the rotation distance between binary trees. *Adv. Math.*, **223**, 1316–1355.

[18] Lucas, J. (2004) Untangling binary trees via rotations. *Comp. J.*, **47**, 259–269.