Time Complexity Analysis of the Binary Tree Roll Algorithm

Adrijan Božinovski[1], George Tanev[1], Biljana Stojčevska[1], Veno Pačovski[1], Nevena Ackovska[2]

[1]School of Computer Science and Information Technology, University American College Skopje, Macedonia

[2]Faculty of Computer Science and Engineering, University "Sv. Kiril i Metodij", Skopje, Macedonia

bozinovski@uacs.edu.mk, george.tanev@uacs.edu.mk, stojcevska@uacs.edu.mk, pachovski@uacs.edu.mk, nevena.ackovska@finki.ukim.mk

Contribution to the state of the art

DOI: **10.7251/JIT1602053B** UDC: **519.857:004.021**

Abstract: This paper presents the time complexity analysis of the Binary Tree Roll algorithm. The time complexity is analyzed theoretically and the results are then confirmed empirically. The theoretical analysis consists of finding recurrence relations for the time complexity, and solving them using various methods. The empirical analysis consists of exhaustively testing all trees with given numbers of nodes and counting the minimum and maximum steps necessary to complete the roll algorithm. The time complexity is shown, both theoretically and empirically, to be linear in the best case and quadratic in the worst case, whereas its average case is shown to be dominantly linear for trees with a relatively small number of nodes and dominantly quadratic otherwise.

Keywords: Binary Tree Roll Algorithm, time complexity, theoretical analysis, empirical analysis.

Introduction

Binary Tree Roll is an operation by which all of the nodes of a binary tree are rearranged in such a way, so that two of the depth-first traversals of the newly obtained binary tree yield the same results as other two traversals of the original binary tree. The graphical representation of the newly obtained binary tree is that it appears to be rolled at a 90 degree angle (either counterclockwise or clockwise, depending on the direction of the applied roll operation) relative to the original binary tree; hence the name "Binary Tree Roll".

This operation was introduced and defined in [1]. There are two variants of the Binary Tree Roll Operation: a counterclockwise (CCW) and a clockwise (CW) roll. The counterclockwise roll of a binary tree, abbreviated as `CCW()`, is defined as follows. Given two binary trees T_1 and T_2, as well as their respective `preorder()`, `inorder()` and `postorder()` traversal functions, operation `CCW()` is defined as in Definition 1:

$$CCW(T_1) = T_2 \Leftrightarrow (preorder(T_1) = inorder(T_2) \wedge inorder(T_1) = postorder(T_2)) \quad (1)$$

In other words, upon `CCW()`, the preorder traversal of the original tree is identical to the inorder traversal of the tree obtained by the counterclockwise roll, and the inorder traversal of the original tree is identical to the postorder traversal of the tree obtained by the counterclockwise roll.

Likewise, the clockwise roll of a binary tree, abbreviated as `CW()`, is defined as in Definition 2:

$$CW(T_1) = T_2 \Leftrightarrow (inorder(T_1) = preorder(T_2) \wedge postorder(T_1) = inorder(T_2)) \quad (2)$$

Similarly, upon `CW()`, the inorder traversal of the original tree is identical to the preorder traversal of the tree obtained by the clockwise roll, and the postorder traversal of the original tree is identical to the inorder traversal of the tree obtained by the clockwise roll.

A graphical explanation was given in [1], showing how the resulting binary tree is obtained visually, so as to comply with definition (1) or (2), depending on the direction of the roll. The downshift visual operation, illustrated in Figure 1, was also presented. It was shown that `CCW()` and `CW()` are inverses of each other, and algorithms for `CCW()` and `CW()` were given, which didn't require obtaining the traversals of the input tree in order to generate the rolled tree.

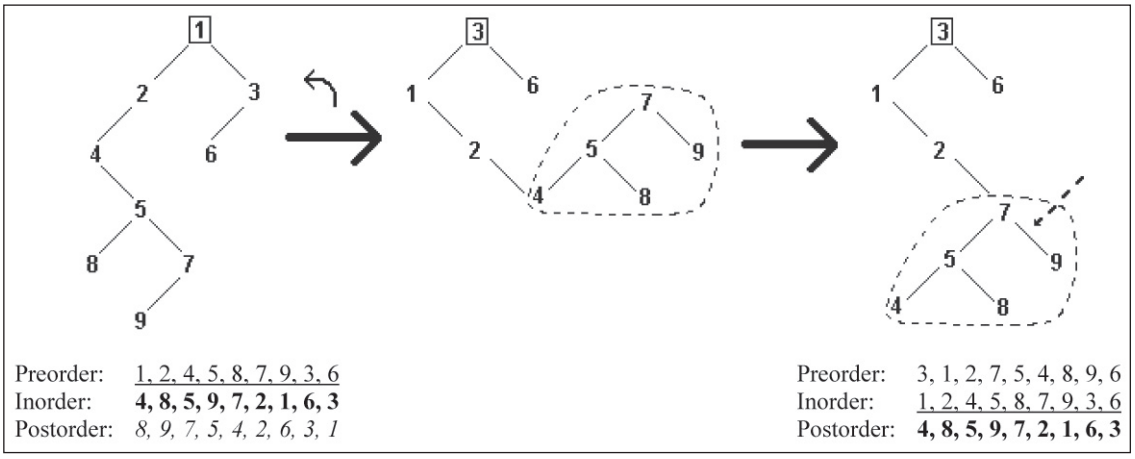

Figure 1. Graphical explanation of the `CCW()` algorithm, and an example of a downshift [1]

Structurally, the algorithm presented in [1] contains a trivial case, two basic cases, and a third, more complex one. The pseudocode for both the `CCW()` and `CW()` variations of the algorithm are shown in Figure 2.

The algorithm takes two input parameters, which represent two binary tree nodes: the `root` of the tree to be `processed`, and its `predecessor`. The predecessor's initial value is always `NULL`, since the root of the input tree never has a predecessor node. However, the value of the `predecessor` parameter changes as further recursive calls to the algorithm are being invoked from inside the function itself. Moreover, the values of both the `root` and the `predecessor` nodes are guaranteed to change within subsequent recursive function calls, since the entire structure of the binary tree is rearranged after the roll operation executes fully.

The motivation for this paper was the fact that the binary tree roll algorithm, in either its `CCW()` or `CW()` variant, has not been analyzed for time complexity so far. That is the goal of this paper and it will be done as follows, focusing on the `CCW()` variant. First, a theoretical analysis of the time complexity will be given, treating all cases of the algorithm execution. Recurrence relations for the time complexity will be stated and proved using mathematical tools. Afterwards, it will be shown how those results are tested empirically, addressing the analytical results for the time complexities of the worst case, best case and average case of the algorithm. Finally, the paper will end with a conclusion about the material presented herein.

```
1.  CCW(&root, &predecessor)
2.  {
3.    if(root != NULL)
4.    {
5.      if(root.rSn == NULL)
6.      {
7.        root.rSn = root.lSn;
8.        root.lSn = NULL;
9.        CCW(root.rSn, root);
10.     }
11.     else
12.     {
13.       if(root.rSn.rSn == NULL)
14.       {
15.         root.rSn.rSn = root.rSn.lSn;
16.         root.rSn.lSn = root;
17.         root = root.rSn;
18.         root.lSn.rSn = root.lSn.lSn;
19.         root.lSn.lSn = NULL;
20.         if(predecessor != NULL)
21.             predecessor.rSn = root;
22.         CCW(root.lSn.rSn, root.lSn);
23.         CCW(root.rSn, root);
24.       }
25.       else
26.       {
27.         CCW(root.rSn, root);
28.         define leftmost = root.rSn;
29.         while(leftmost.lSn != NULL)
30.             leftmost = leftmost.lSn;
31.         leftmost.lSn = root;
32.         define newroot = root.rSn;
33.         root.rSn = NULL;
34.         root = newroot;
35.         if(predecessor != NULL)
36.             predecessor.rSn = root;
37.         CCW(leftmost.lSn, leftmost);
38.       }
39.     }
40.   }
41. }
```

a)

```
1.  CW(&root, &predecessor)
2.  {
3.    if(root != NULL)
4.    {
5.      if(root.lSn == NULL)
6.      {
7.        root.lSn = root.rSn;
8.        root.rSn = NULL;
9.        CW(root.lSn, root);
10.     }
11.     else
12.     {
13.       if(root.lSn.lSn == NULL)
14.       {
15.         root.lSn.lSn = root.lSn.rSn;
16.         root.lSn.rSn = root;
17.         root = root.lSn;
18.         root.rSn.lSn = root.rSn.rSn;
19.         root.rSn.rSn = NULL;
20.         if(predecessor != NULL)
21.             predecessor.lSn = root;
22.         CW(root.rSn.lSn, root.rSn);
23.         CW(root.lSn, root);
24.       }
25.       else
26.       {
27.         CW(root.lSn, root);
28.         define rightmost = root.lSn;
29.         while(rightmost.rSn != NULL)
30.             rightmost = rightmost.rSn;
31.         rightmost.rSn = root;
32.         define newroot = root.lSn;
33.         root.lSn = NULL;
34.         root = newroot;
35.         if(predecessor != NULL)
36.             predecessor.lSn = root;
37.         CW(rightmost.rSn, rightmost);
38.       }
39.     }
40.   }
41. }
```

b)

Figure 2. The algorithms for a) CCW() and b) CW()[1]

Time Complexity – Analytical Approach

Depending on the topology of the tree being processed by the roll algorithm, any one of the three cases is equally likely to be invoked (the ones initiated with lines 5, 13 and 25 in Figure 2, respectively). Therefore, the time complexity equation can be written as in Equation 3:

$$T(n) = \begin{cases} T_0(n) \\ T_I(n) \\ T_{II}(n) \\ T_{III}(n) \end{cases} \quad (3)$$

where $T_0(n)$, $T_I(n)$, $T_{II}(n)$, and $T_{III}(n)$ denote the trivial, first, second, and third case of the algorithm, respectively. The trivial case (line 3 in Figure 2) is always executed in constant time and produces no further recursive calls, yielding $T_0 = \Theta(1)$. The three non-trivial cases will be analyzed with the assumption that the trivial case is fulfilled. Furthermore, the number of nodes of the tree will be denoted by n, the line numbers will refer to the algorithm in Figure 2, and the recurrences obtained will be followed by the results obtained by solving those recurrences using the backward substitution method [4], or by the substitution method based on mathematical induction [3].

The analysis will be done upon the `CCW()` version of the algorithm, i.e. it will concern Figure 2a. As stated in [1], the `CW()` algorithm is an inverse of `CCW()`; substituting "left" for "right" and vice versa, as well as `CCW()` for `CW()` (for the recursive calls), will transform the `CCW()` algorithm into the `CW()` algorithm, so the following analysis can thus be used for the `CW()` variant as well.

First case - lines 5-10

This case is invoked when the root has no right sub-tree (line 5). The following lines of code take the root's left sub-tree and make it its right sub-tree. Then, a recursive call is made on the new right sub-tree. Figure 3 presents this case visually.

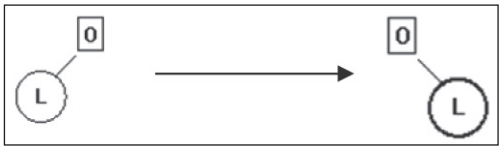

Figure 3. The first basic case in the `CCW()` algorithm [1]

This case will yield a constant number of operations (two), as well as a recursive call invoked upon a tree containing $n - 1$ nodes (i.e. the root's only sub-tree). Thus, the time complexity recurrence for the first case can be written as in Equation 4:

$$T_I(n) = 2c + T(n-1) \tag{4}$$

where the $T(n-1)$ recursive call implies that any one of the cases of the algorithm may be invoked, depending on the topology of the remainder of the tree. Solving the recurrence results in a tightly linear complexity as stated in Equation 5.

$$T_I(n) = \Theta(n) \tag{5}$$

Second case - lines 11-24

This case is activated when the root of the tree has a right sub-tree, which does not have a right sub-tree of its own (line 13). There will be 6 or 7 operations needed to roll the root, its right child node, and their respective left sub-trees counterclockwise (6 if the root of the initial tree is rolled using this case, 7 if any sub-tree is rolled using this case), and two additional recursive calls on the two formerly left and ultimately right sub-trees. The number of constant operations can be denoted as Kc (where K is either 6 or 7) and will be included in the time complexity equation for the second case, shown visually in Figure 4.

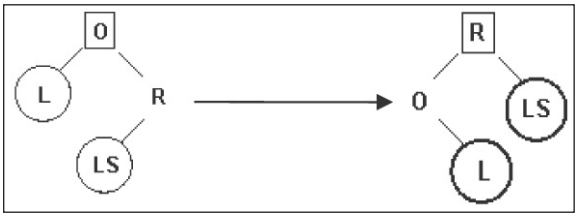

Figure 4. The second basic case in the `CCW()` algorithm [1]

There are two recursive calls, so it is necessary to determine the extreme scenarios upon which they can potentially be invoked. In the worst-case scenario, all of the remaining nodes will be in one of the sub-trees, whereas the other one will remain empty. This scenario can be described as in Equation 6:

$$T_{II_0}(n) = Kc + T(n-2) + T(0) \tag{6}$$

where the $T(n-2)$ denotes that the two nodes already rolled with the constant number of lines (lines 15 to 20 or 21) will not be worked with again. Solving this recurrence gives a result of a tight linear complexity, as shown in Equation 7:

$$T_{II_0}(n) = \Theta(n) \tag{7}$$

The best-case scenario is when both of the sub-trees have an equal number of the remaining $n - 2$ nodes. This can be represented as in Equation 8:

$$T_{II_\Omega}(n) = Kc + 2T\left(\frac{n-2}{2}\right) \quad (8)$$

Solving this recurrence again yields a solution of tight linear complexity. This is shown in Equation 9:

$$T_{II_\Omega}(n) = \Theta(n) \quad (9)$$

Since both extreme scenarios of the second case have linear time complexities, it follows that the second case of the binary tree roll algorithm has tightly linear time complexity (Equation 10):

$$T_{II}(n) = \Theta(n) \quad (10)$$

Third case - lines 25-38

This case gets invoked when the root has a right sub-tree, which has a right sub-tree of its own. As stated in [1], this case deals with the downshift of stems of right child nodes and transforming them into stems of left child nodes. The algorithm first creates a recursive call upon the right sub-tree of the root and it continues to do so until a basic case is reached (i.e. until a sub-tree with at most one right child node is reached, following the stem of right child nodes from the root towards its rightmost child node). When such a case is handled by the algorithm, the remainder of the third case relocates the former root of the tree to be the "leftmost" child node in the newly rolled tree, and the procedure is then recursively invoked again on the former root (and its entire left sub-tree), now placed as the leftmost node in the sub-tree handled by the third case. Figure 5 shows the third case visually.

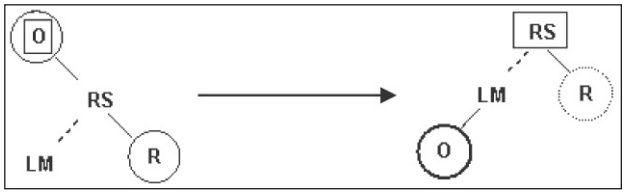

Figure 5. The third and most complex case in the `CCW()` algorithm [1]

How many times the loop in case 3 (lines 29 and 30) will be executed depends on the number of nodes in the stem of right child nodes of the root of the tree on which the `CCW()` algorithm is invoked, if the third case of the algorithm applies to it. After downshifting, this stem will become a stem of left child nodes, and the root node will need to be linked as the left child node of the leftmost node in this stem. Since the rightmost node in the original tree will become the root of the new tree after `CCW()` is performed on it, finding the leftmost node will need to be done from the new root towards the left, which is the reason for the loop in lines 29 and 30 of the algorithm.

To help quantify this and precisely determine the time complexity of the third case of the algorithm, additional variables can be introduced. These are presented visually in Figure 6.

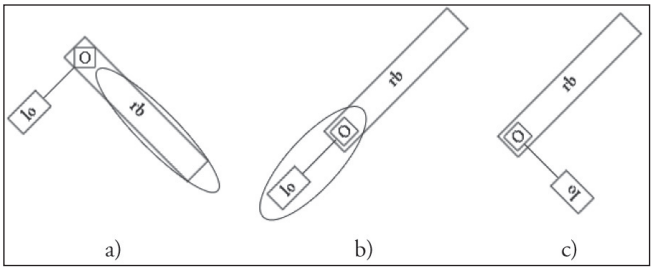

Figure 6. The third case of the `CCW()` algorithm: a) the head recursion (ellipse) of the third case deals with the stem of right child nodes ( ) and transforms it into a stem of left child nodes via downshift; b) the root ( ) is linked as the leftmost in the stem of left child nodes and the tail recursion (ellipse) of the third case is invoked upon it; c) since the former root does not have a right child node of its own, the tail recursion will invoke the first case, and the left sub-tree of the former root ( ) will become its right sub-tree

As shown in Figure 6, *rb* represents the number of nodes in the stem of right child nodes of the root of the tree (including it), whereas *lo* is the number of nodes in the left sub-tree of the root. Having this in mind, the time complexity recurrence of the third case can be written as in Equation 11:

$$T_{III}(n) = T(n-1-lo) + 2c + 2c(rb-2) + Pc + T(1+lo) \quad (11)$$

where Constraints 12 and 13 apply:

$$3 \le rb \le n \quad (12)$$

$$0 \le lo \le n - rb \quad (13)$$

The five terms of the Equation 9 and the constraints in Equations 10 and 11 are explained as follows:

- $T(n - 1 - lo)$ is the head recursion of the third case (line 27). It will be invoked upon all of the nodes in the tree (n), except the root and its left sub-tree (lo), as shown in Figure 6a;
- $2c$ is the constant time needed to invoke the two lines of code 28 and 29;
- 2c(rb – 2) is the time needed to invoke the loop in lines 30 and 29 again (the loop condition test). The two lines of code in the loop will be invoked for all nodes in the stem of right child nodes (turned into a stem of left child nodes after the downshift, i.e., after the head recursion has completed) except for two: the root (the head recursion, i.e., downshift, is invoked for the right child node of the root, meaning that the root does not become downshifted until all other nodes in the stem of right child nodes get downshifted) and the last node in the stem of right child nodes (because when the head recursion reaches the node which is a parent to the last node in the stem of right child nodes, the second basic case of the `CCW()` will be invoked and not the third case, thus initiating the end of the downshift process). That is why this term is $2c(rb - 2)$.

The third case of the algorithm will be invoked only if the stem of right child nodes contains 3 or more nodes (which can be inferred by consecutively following the tests in lines 3, 5, 13 and 25); if it contains only 3 nodes, that is the best-case scenario, whereas if it contains all n nodes of the tree (i.e., the tree is right-degenerated), that is the worst-case scenario; hence Constraint 12;

- P is a constant which is either 5 or 6, depending on whether the third case of the algorithm is invoked upon the root of the tree (i.e., the node having no predecessor) or any other child node of the tree respectively—this signifies inclusion of lines 31 to 35 and 36, respectively;
- $T(1 + lo)$ represents the tail recursion, which is invoked on the former root of the tree (eventually placed as the leftmost node in the stem of downshifted left child nodes) and the nodes in its left sub-tree (lo), as shown in Figure 6b. In the best-case scenario, this left sub-tree will be empty, and in the worst-case scenario it will contain all the nodes of the tree except the ones contained in the initial stem of right child nodes (which includes the root of the tree as well, as shown in Figure 6), from where Constraint 13 is derived.

In order to analyze Equation 11, the extreme scenarios for Constraints 12 and 13 need to be addressed. The worst-case scenario in Constraint 12 is when $rb = n$, i.e., when the stem of right child nodes contains all of the nodes of the tree (which means that the tree is right-degenerated). This means that there is no left sub-tree to the root, i.e., $lo = 0$, which can be inferred both logically and formally, by substituting $rb = n$ in Constraint 13. Thus, Equation 11 becomes Equation 14:

$$T_{III_0}(n) = T(n - 1) + 2c + 2c(n - 2) + Pc + T(1) \quad (14)$$

Solving this recurrence yields a tightly quadratic complexity, as stated in Equation 15:

$$T_{III_0}(n) = \Theta(n^2) \quad (15)$$

The best-case scenario occurs when the third case of the algorithm is invoked only once for the entire tree, i.e., when $rb = 3$ in Constraint 12. Substituting this in Equation 11 produces Equation 16:

$$T_{III_\Omega}(n) = T(n - 1 - lo) + 2c + 2c + Pc + T(1 + lo) \quad (16)$$

There are two extremes of the best-case scenario to consider:

- The first extreme is when $lo = 0$ in the constraint in Equation 11, which means that there is no left sub-tree to the root. Then, the head recursion would handle all of the nodes of the tree except the root (which are not in the left sub-tree of the root and do not form a stem of right child nodes), whereas the tail recursion would handle only the root; this can be inferred by substituting the aforementioned value for lo in Equation 16 and thus obtaining Equation 17:

$$T_{III_{\Omega_1}}(n) = T(n - 1) + 2c + 2c + Pc + T(1) \quad (17)$$

Solving the recurrence in Equation 17 results in a tightly linear complexity (since $T(1) = \Theta(1)$, i.e. it is a constant), as stated in Equation 18:

$$T_{III_{\Omega_1}}(n) = \Theta(n) \quad (18)$$

- The second extreme is when $lo = n - 3$ in Constraint 13, which means that the left sub-tree of the root contains all of the nodes of the tree, ex-

cept the three nodes (including the root) placed in the stem of right child nodes, which would invoke the third case of the algorithm. Thus, the head recursion would handle only two nodes (the lower two nodes of the stem of three right child nodes, handled by the second case), whereas the tail recursion would handle the remaining $n - 2$ nodes of the tree; this can also be inferred by substituting the aforementioned value for *lo* in Equation 16 and thus produce the recurrence in Equation 19:

$$T_{III_{\Omega 2}}(n) = T(2) + 2c + 2c + Pc + T(n-2) \quad (19)$$

Solving this recurrence again results in a tightly linear complexity (since, again, $T(2) = \Theta(1)$, i.e. it is also a constant), as stated in Equation 20:

$$T_{III_{\Omega 2}}(n) = \Theta(n) \quad (20)$$

Thus, since both extremes of the best-case scenario for the third case have linear time complexities, the best-case scenario for the third case, as a whole, has linear time complexity, as stated in Equation 21:

$$T_{III_{\Omega}}(n) = \Theta(n) \quad (21)$$

Comparing Equations 15 and 21, it can be seen that the third case of the algorithm is not robust, i.e. that its time complexity can range from quadratic in the worst case to linear in the best case. Assuming that all of the aforementioned cases of the algorithm (consisting of their worst- and best-case scenarios, including the extreme sub-variants) are equally likely to be invoked, one can undertake a probabilistic approach to the time complexity analysis. More specifically, the complexities of the following cases need to be considered: $T_I(n)$ (Equation 5), $T_{II_0}(n)$ (Equation 7), $T_{II_\Omega}(n)$ (Equation 9), $T_{III_0}(n)$ (Equation 15), $T_{III_{\Omega 1}}(n)$ (Equation 18) and $T_{III_{\Omega 2}}(n)$ (Equation 20). It can be seen that, out of the six possible extreme cases that can arise during the processing of a random binary tree with the `CCW()` algorithm, only one is quadratic and all the others are linear. In other words, it can be assumed that, whenever a random tree is processed by the `CCW()` algorithm, five times out of six the algorithm will have a linear time complexity, and once out of six times it will have a quadratic time complexity.

Of course, this is a simplified model of the time complexity of the algorithm, and further research into the topic is needed. Specifically, experimental research needs to be conducted, where the number of steps to execute the algorithm needs to be counted for certain topologies of the binary tree structure, in order to obtain a clear picture into the time complexity of the algorithm.

Time Complexity – Empirical Approach

It is reasonable to expect that not all topologies of binary trees with certain amounts of nodes will require the same amounts of time to have the binary roll operation fully performed on them. Figure 7 presents all possible topologies of binary trees for $n = 3$ and $n = 4$. For every binary tree with n nodes there are C_n possible binary tree topologies, where C_n is the n-th Catalan number.

Based on the equations for time complexity, a logical assumption would be that, for a tree with $n = 3$ nodes, one out of $C_3 = 5$ trees would need quadratic time complexity to have CCW roll performed on it (Figure 7a) — it would be the one containing three nodes in the stem of right child nodes, i.e. the topology of a right-degenerated tree. Also, for a tree with $n = 4$ nodes, four out of $C_4 = 14$ trees would need quadratic time complexity to have CCW roll performed on them (Figure 7b) — the ones that have stems of three and more right child nodes.

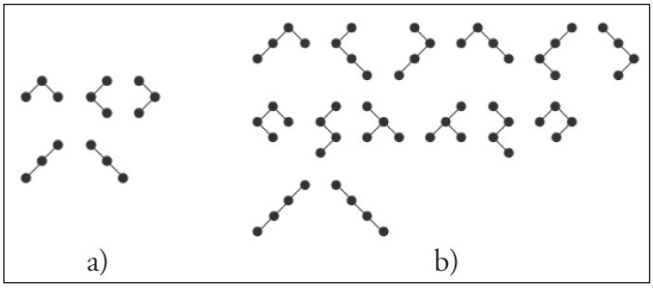

Figure 7. The topologies of binary trees for a) $n = 3$ and b) $n = 4$

In order to be certain about how much time is needed to perform CCW roll on a tree with n nodes, an exhaustive analysis needs to be performed. This includes obtaining all topologies of binary trees with n nodes and performing CCW roll on all of them, while counting the steps (i.e., time units) until the CCW roll completes. For this, it is necessary to first generate all topologies of binary trees for a

given n and then execute CCW roll on all of them, while counting the steps during the CCW roll executions. Following the theoretical analysis, it is expected that there will be a quadratic time complexity for the worst case and a linear time complexity for the best case, and it is therefore necessary to collect information for both. It is also appealing to know whether the time complexity of the algorithm would be more dominantly linear or quadratic, i.e., whether the best case or worst case of the algorithm would be more likely to be invoked during the execution of the CCW roll. For this reason, an average time complexity would also be extracted, as an average of the time complexities for all topologies of binary trees for a given number of nodes n.

In order to obtain all topologies of binary trees with a given number of nodes, the Catalan Cipher Vector approach is used in this paper. A Catalan Cipher Vector [2] is a vector which uniquely determines a binary tree's topology. For a tree with n nodes, there will be C_n topologies of binary trees and thus C_n Catalan Cipher Vectors. Table 1 shows all the ranks, their corresponding Catalan Cipher Vectors, and the appropriate binary trees, for $n = 4$ nodes.

Since the initial Catalan Cipher Vector for a tree with n nodes is always $[0\ 1\ 2\ \ldots\ n-1]$ [2], it is possible to generate the corresponding binary tree for it, and count the number of steps it would take to complete the `CCW()` on it. Then, the subsequent Catalan Cipher Vector can be obtained, the corresponding binary tree can be generated from it, have `CCW()` executed on it and count the number of steps needed and so on, until all C_n binary tree topologies are processed this way. Throughout the process, the maximum and the minimum number of steps needed are tracked, as well as the total number of steps for all the C_n topologies of the binary trees with n nodes. These can be plotted on a graph for different subsequent values of n, in order to provide a graphical representation of the best case, worst case and average case of the time complexity of the `CCW()` algorithm, respectively.

The results for such an analysis have been performed and the results are given in Table 2.

Table 1. Ranks and enumerations of the binary trees with $n = 4$ nodes using the Catalan Cipher Vector approach

Rank	Catalan Cipher Vector	Binary Tree
0	[0 1 2 3]	
1	[0 1 2 4]	
2	[0 1 2 5]	
3	[0 1 2 6]	
4	[0 1 3 4]	
5	[0 1 3 5]	
6	[0 1 3 6]	
7	[0 1 4 5]	
8	[0 1 4 6]	
9	[0 2 3 4]	
10	[0 2 3 5]	
11	[0 2 3 6]	
12	[0 2 4 5]	
13	[0 2 4 6]	

Table 2. Numbers of steps necessary to perform `CCW()` on all topologies of binary trees with given numbers of nodes

n	$C(n)$	*Min*	*Max*	*Avg*	*Total*
2	2	9	11	10	20
3	5	13	29	18	88
4	14	17	49	26	360
5	42	21	71	34	1.430
6	132	25	95	43	5.610
7	429	29	121	51	21.890
8	1.430	33	149	60	85.228
9	4.862	37	179	68	331.630
10	16.796	41	211	77	1.290.640
11	58.786	45	245	85	5.025.880
12	208.012	49	281	94	19.586.720
13	742.900	53	319	103	76.399.836
14	2.674.440	57	359	112	298.274.350
15	9.694.845	61	401	120	1.165.544.550
16	35.357.670	65	445	129	4.558.478.100
17	129.644.790	69	491	138	17.843.217.150
18	477.638.700	73	539	146	69.899.012.040
19	1.767.263.190	77	589	155	274.028.145.600
20	6.564.120.420	81	641	164	1.075.046.854.800

The results are interpreted as follows. In the first data row, for a tree with $n = 2$ nodes (first column), there are $C(n) = 2$ (second column) total topologies of binary trees. Executing `CCW()` on all of them yields a *Total* (sixth, i.e. last column) of 20 time units, leading to an *Avg* (average – fifth column) of 10 time units per binary tree topology. Of all topologies, the *Min* (minimum – third column) number of time units necessary to complete `CCW()` on a binary tree topology with 2 nodes is 9, and the *Max* (maximum – fourth column) number of such time units is 11. This interpretation follows all rows of the table, up to and including binary tree topologies for $n = 20$ nodes.

Plotting the obtained data results in a chart like in Figure 8.

The results concur with the theoretical analysis: the algorithm has a quadratic time complexity in the worst case and a linear time complexity in the best case, whereas the average case has a near-linear complexity. It is possible to interpolate precise equations from the data for the worst and best case, and these are $T_{max}(n) = n^2 + 13n - 19$ and $T_{min}(n) = 4n + 1$, respectively.

For the average case, the most accurate interpolation is a quadratic one, since the plot indicates such a complexity, which is also confirmed using the least square error method. The equation thus obtained is $T_{avg}(n) = 0.0129n^2 + 8.3308n + 0.8554$. This shows that the quadratic term of the function will be shadowed by the linear term for smaller values of n, but that it would eventually become dominant when n grows beyond a certain threshold. The threshold would be the value of n for which the quadratic term becomes larger than the linear term, or the value of n for which $0.0129n^2 > 8.3308n$. This is true for $n > \frac{8.3308}{0.0129} \approx 645{,}798 \geq 646$, since n is an integer. In other words, for trees up to 645 nodes there is a higher probability that the `CCW()` will perform a binary tree roll in linear time, whereas for trees with 646 and more nodes there is a higher probability that the `CCW()` will perform binary tree roll in quadratic time.

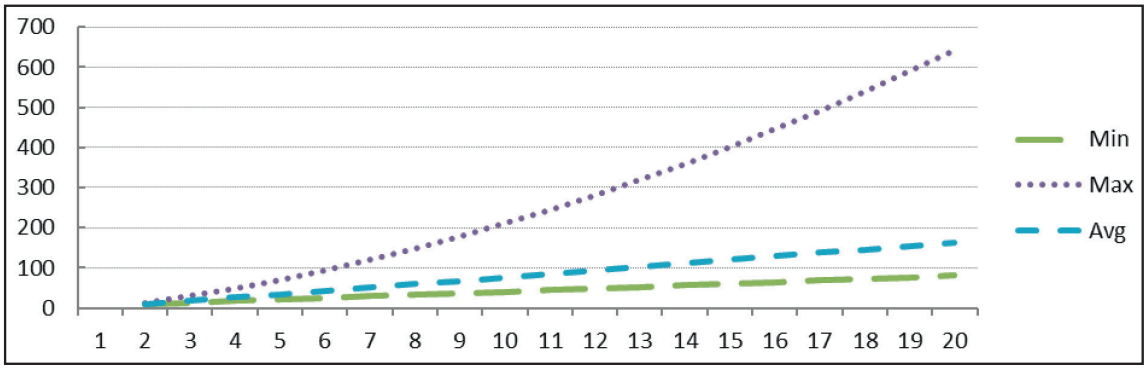

Figure 8. A plot of the results given in Table 2

CONCLUSION

This paper presented an analysis of the time complexity of the binary tree roll algorithm, specifically its counterclockwise (`CCW()`) variant, with the note that the analysis for its clockwise (`CW()`) variant is analogous. For the time complexity analysis, the trivial and the three non-trivial cases of the algorithm were presented and recurrence relations for them were derived and solved. The results from the theoretical analysis were checked empirically, by performing exhaustive testing on all trees with given numbers of nodes n, counting all the steps while performing the algorithm. The theoretical results, that the time complexity of the `CCW()` algorithm is linear in the best case and quadratic in the worst case, were confirmed by the empirical results. Furthermore, the average case analysis showed that the `CCW()` algorithm is dominantly linear for trees with $n \leq 645$, whereas for trees with higher numbers of nodes the quadratic time complexity becomes more dominant.

BIOGRAPHY

Adrijan Božinovski works as an Associate Professor at the School of Computer Science and Information Technology at University American College Skopje, where he is currently the Dean. He obtained his BSc from University "St. Cyril and Methodius" in Skopje, Macedonia, and his MSc and PhD from University of Zagreb, Croatia.

George Tanev is an MSc graduate student of the School of Computer Science and Information Technology at University American College Skopje, Macedonia, where he acquired his BSc in Computer Science. Also works as a software developer in Skopje, Macedonia.

Biljana Stojčevska works as an Associate Professor at the UACS School of Computer Science and Information Technology. She received her BSc, MSc and PhD degrees in Computer Science at the Institute of Informatics, Faculty of Natural Sciences and Mathematics, at "Sts. Cyril and Methodius University" in Skopje, Macedonia.

Veno Pachovski (1965) graduated, completed MSc and got his PhD from Faculty of Natural Sciences and Mathematics, University "Sts. Cyril And Methodius", Skopje, Macedonia.

Since 2009, he teaches a variety of courses at the University American College – Skopje, mainly within the School of Computer Sciences and Information technology (SCSIT).

Nevena Ackovska is Associate Professor at the Faculty of Computer Science and Engineering at "St. Cyril and Methodius" University in Skopje, Macedonia. She holds B.Sc. in Computer Engineering, Informatics and Automation from Electrical Engineering Faculty (2000), M.Sc. in Bioinformatics (2003) and a Ph.D. in Bioinformatics (2008) from Faculty of Natural Sciences and Mathematics at "St. Cyril and Methodius University" in Skopje, Macedonia.

REFERENCES:

[1] Božinovski, A. and Ackovska, N. (2012) The Binary Tree Roll Operation: Definition, Explanation and Algorithm, International Journal of Computer Applications, 46(8):40-47.

[2] Božinovski, A., Stojčevska, B. and Pačovski, V. (2013) Enumeration, Ranking and Generation of Binary Trees Based on Level-Order Traversal Using Catalan Cipher Vectors, Journal of Information Technology and Applications, 3(2):78-86.

[3] Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C. (2009) *Introduction to Algorithms, Third Edition*, The MIT Press.

[4] Puntambekar, A. A. (2010) *Design and Analysis of Algorithms*, Technical Publications Pune.

Submitted: December 1, 2016.
Accepted: December 12, 2016.