Evaluating NAND Trees

Group

Adarsh Parihar - 17IE10035
Rounak Garg - 17IE10041
Aditya Basu - 19IE10002

Course: Algorithms-1 (CS21003)
Guidance: Prof. Palash Dey and Prof. Partha P Chakrabarti

Contents

Major Problem Statement

- “Designing, analysing and implementing fast algorithms to evaluate NAND Trees”

- The major problem statement has three subparts.

Motivation

NAND trees and their evaluation are used to design and improve several real systems:

1. Developing faster signal evaluation algorithms of the nature of evaluation of NAND trees are used to make burglar alarms more effective and robust.

2. NAND Trees offer standardized facility for hardware fault detection. They are used in various semiconductor test methods. For example, the "NAND Tree test" is a commonly used test.

Quick recap - NAND Gate

The operation: X = A NAND B = !(A && B) i.e, NOT(A AND B)

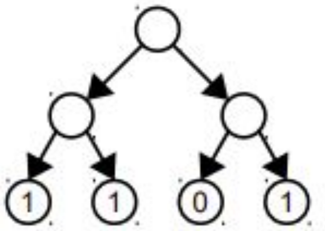
The gate

Truth Table

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

What is a NAND Tree ?

A NAND tree is a complete binary tree with the following properties:

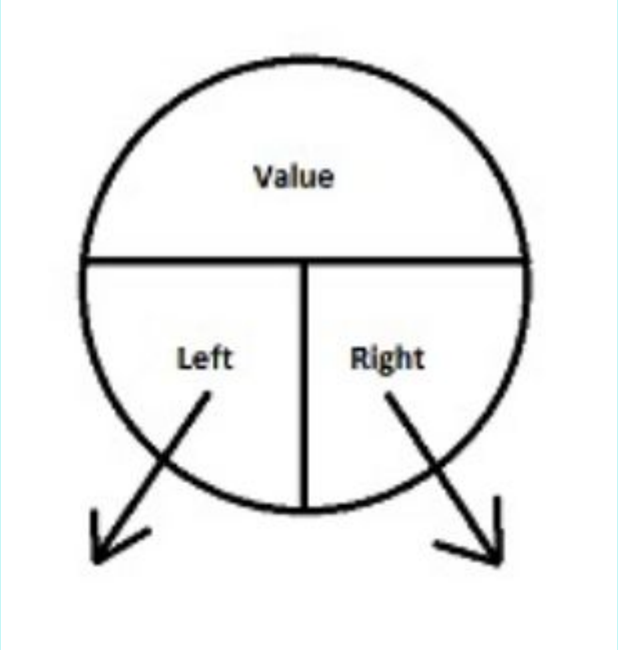


- Each leaf node is labelled either 0 or 1
- All internal nodes are NAND gates. A NAND gate is a logic gate that takes in two inputs and evaluates to 0 if both its inputs are 1 and to 1 if either input is 0.
- Essentially, this means that for any internal node:

  node.value = (node.left_child) NAND (node.right_child)

Structure of a node in a NAND tree

```
struct node
{
	bool value;
	struct node *left;
	struct node *right;
};
```



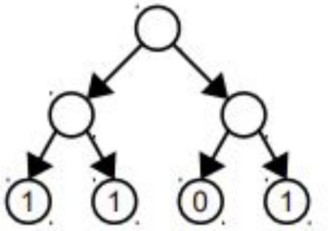
Naive recursive algorithm to evaluate a NAND tree

Evaluation of NAND Tree: Finding the value of top-level NAND gate of the tree (value of the root)

Pseudocode:

```
bool naive_eval_NAND (node *root)
{
    if (root.left = NULL)  // tree is a single leaf node
        return root

    l = naive_eval_NAND (root.left)
    r = naive_eval_NAND (root.right)
    return (l NAND r)
}
```



This NAND tree evaluates to 1!

Naive recursive algorithm to evaluate a NAND tree

Pseudocode:

```
bool naive_eval_NAND (node *root)
{
    if (root.left = NULL)
        return root.val

    l = naive_eval_NAND (root.left)
    r = naive_eval_NAND (root.right)
    return (l NAND r)
}
```

Analysis:

Let n be the # nodes in the tree

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ 2T(n/2) + 1, & \text{otherwise} \end{cases}$$

Thus, using Master's theorem:
$T(n) = \Theta(n)$

Note: Even if #leaves = n, we'd get the same running time since:
#nodes = 2 #leaves - 1

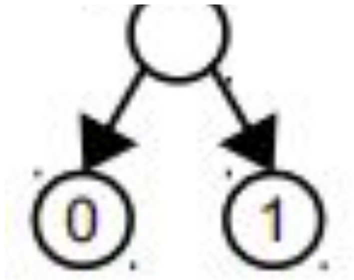
Short-circuiting - Improving the naive Θ(n) algorithm

- We exploit a particular property of NAND gates to improve our naive algorithm:

  0 NAND X = 1 i.e 0 NAND 0 = 1, 0 NAND 1 = 1

- While evaluating a node, if one of its subtree evaluates to 0, then we don't need to evaluate the other subtree.

- We use the above property. This is referred to as the "short-circuiting" algorithm.



In this case,
Don't need to evaluate right subtree since left subtree evaluates to 0 !
Hence, NAND tree evaluates to 1

“Left-first” - A deterministic short-circuiting algorithm

- While evaluating a node, always evaluate the **<u>left subtree of the node first</u>**.

- If the left subtree evaluates to 0, we’re done, we return 1.
  (i.e. short-circuiting successful!)

- If the left subtree evaluates to 1, only then evaluate the right subtree of the node.
  Then take the NAND operation.

Pseudocode:

```
bool left_first (node *root){
    if (root.left = NULL)  // leaf
    return root.val

    l = left_first (root.left)
    if( l = 0) return 1
    else {
    r = left_first (root.right)
    return (l NAND r)
    }
}
```

“Left-first” - A deterministic short-circuiting algorithm

- The left-first algorithm definitely works at least as good as our naive recursive algorithm in all cases.
- In many cases, left-first works faster than the naive algorithm (when it is able to successfully short-circuit many times!)
- However, it is possible to design a NAND tree where left-first wouldn’t short-circuit even once!
- This is where we come to the first part of our problem statement.

Designing the worst case!

Designing the worst case of "left-first"

Upshot of the first part of the problem statement:

- In many cases, left-first works faster than the naive recursive algorithm (when it is able to successfully short-circuit many times!)
- We design, analyze and implement an algorithm to create a NAND tree on which left-first wouldn't short-circuit even once!
- This is designing the worst case. Consequently, time complexity of left-first would be $\Theta(n)$.

Designing the worst case of "left-first"

High-level logic of the algorithm that creates the worst case:

- The property we have leveraged for short-circuiting 0 NAND X = 1
- To ensure that left-first never short-circuits, we need to ensure that the left subtree of ALL nodes never evaluates to 0
- Thus, while creating a node we always assign its left sub-child the value 1
- The right sub-child is assigned 0 or 1 depending on the value of the node is.
- The algorithm builds the tree from the root to the leaf using recursion.

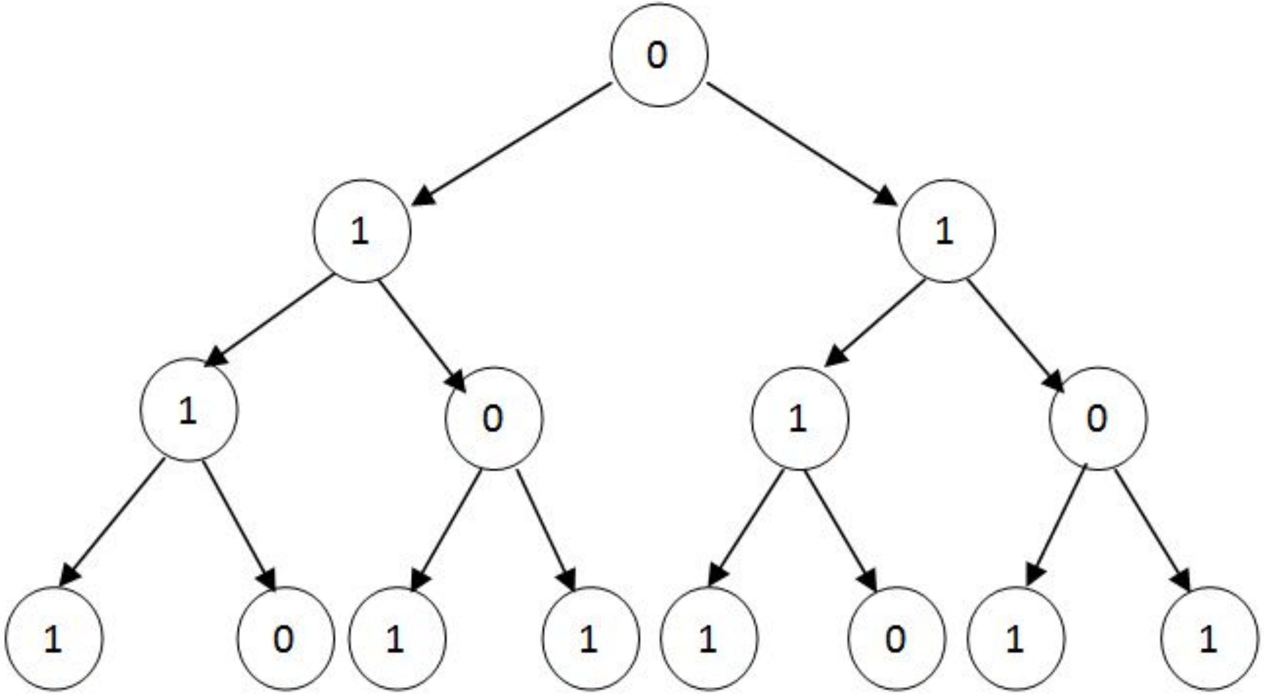
Designing the worst case of "left-first"

Pseudocode:

```
 struct tree* make_tree (bool value, int
k, int i) {
    struct tree* node = new tree();
    // allocate space for the node
    node->val = value;
    node->left = NULL;
    node->right = NULL; // initialization
    if (i == k) return node;
    //current level = required level
    int v =1;
    node->left = make_tree(v, k, i+1);
    if (value == 1) v = 0;
    node->right = make_tree(v, k, i+1);
    return node;
}
```

Designing the worst case of "left-first"

Example:



Designing the worst case of "left-first"

Analysis:

Let k be a number such that # leaves (n) = 2^k

Consequently, # nodes in the tree = 2n - 1

$$T(k) = 1, \text{ if } k = 0$$
$$= 2T(k-1) + 1, \text{ otherwise}$$

Thus using simple recurrence,

$T(k) = \Theta(2^k)$

That is, $T(n) = \Theta(n)$

Thus, we get linear time complexity in building the worst case tree.

Designing a randomized algorithm!

“Random-first” - A randomized short-circuiting algorithm

Upshot of the second part of the problem statement:

- In many cases, left-first works faster than the naive recursive algorithm (when it is able to successfully short-circuit many times!)
- But, we successfully designed a worst case where its running time is $\Theta(n)$
- We try to improve this by designing a randomized algorithm “random-first” which does better than $\Theta(n)$ in expectation (sublinear)

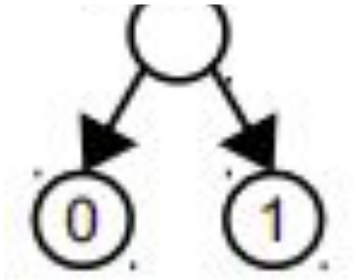
Recap - Short-circuiting

- We exploit a particular property of NAND gates to improve our naive algorithm:

$$0 \text{ NAND } X = 1 \quad \text{i.e } 0 \text{ NAND } 0 = 1, 0 \text{ NAND } 1 = 1$$

- While evaluating a node, if one of its subtree evaluates to 0, then we don't need to evaluate the other subtree.

- We use the above property. This is referred to as the "short-circuiting" algorithm.



In this case,
Don't need to evaluate right subtree since left subtree evaluates to 0 !
Hence, NAND tree evaluates to 1

“Random-first” - A randomized short-circuiting algorithm

- While evaluating a node, we randomly choose a subtree first (either left or right))

- We evaluate the **<u>randomly chosen subtree (either left or right) first.</u>**
- If the randomly chosen subtree evaluates to 0, we’re done, we return 1.
(i.e. short-circuiting successful!)
- If the randomly chosen subtree evaluates to 1, only then evaluate the other subtree of the node.
- Then take the NAND operation and return the value.

"Random-first" - A randomized short-circuiting algorithm

Pseudocode:

```
bool random_first (node *root) {
    if (root.left = NULL)  // at a leaf node
        return root.val
    choice = rand() % 2
    if (choice == 0) {
        l = random_first (root.left)
        if (l = 0) return 1
        else {
            r = random_first (root.right)
            return (l NAND r)
        }
    }
    else {
        r = random_first (root.right)
        if (r = 0) return 1
            else {
            l = random_first (root.left)
            return (l NAND r)
            }
        }
    }
```

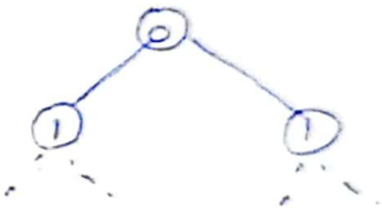
Recurrence Relations for $T_0(n)$ and $T_1(n)$

Let $T_0(n)$ be the expected runtime of the random-first algorithm on a tree with n leaf nodes assuming the root evaluates to 0.

$$\boxed{T_0(1) = \Theta(1)} \Leftarrow \quad \therefore \text{Constant Time}$$

$$T_0(n) = 2T_1\left(\frac{n-1}{2}\right) + \Theta(1)$$

$$\Rightarrow \boxed{T_0(n) \leq 2T_1\left(\frac{n}{2}\right) + \Theta(1)}$$

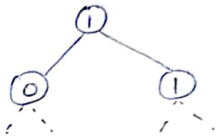


Recurrence Relations for $T_0(n)$ and $T_1(n)$

Let $T_1(n)$ be the expected runtime of the random-first algorithm on a tree with n leaf nodes assuming the root evaluates to 1

$$\boxed{T_1(1) = \Theta(1)}$$ ⇐ CONSTANT TIME

TWO POSSIBLE CASES IN $T_1(n)$



Case 1: Left Subtree (0) is called first

$$T_1(n) \le T_0\left(\frac{n}{2}\right) + \Theta(1)$$

Case 2: Right Subtree (1) is called first

$$T_1(n) \le T_1\left(\frac{n}{2}\right) + T_0\left(\frac{n}{2}\right) + \Theta(1)$$

So $T_1(n)$ [Expected runtime]

$$\le \frac{1}{2}\left[T_0\left(\frac{n}{2}\right) + \Theta(1)\right] + \frac{1}{2}\left[T_1\left(\frac{n}{2}\right) + T_0\left(\frac{n}{2}\right) + \Theta(1)\right]$$

$$\boxed{T_1(n) \le \frac{1}{2} T_1(n/2) + T_0(n/2) + \Theta(1)}$$

Recurrence Relations for $T_0(n)$ and $T_1(n)$

OBSERVATION: $T_1(n) \le T_0(n)$ $\quad T_0(1) = \Theta(1)$

$T_1(1) = \Theta(1)$

TO PROVE: $T_0(n) = O(n^{\varepsilon})$

$\forall \; |\varepsilon < 1|$

USING INDUCTION:

$$T_0\left(\frac{n}{2}\right) = O\left[\left(\frac{n}{2}\right)^{\varepsilon}\right]$$

GIVEN: $T_0(n) \le 2T_1(n/2) + \Theta(1)$

$$T_0(n) \le 2T_0(n/2) + \Theta(1)$$

$$\le 2 \times O\left[\left(\frac{n}{2}\right)^{\varepsilon}\right]$$

$$\le O\left[2^{2k\varepsilon - \varepsilon + 1}\right] \qquad (n = 4^k)$$

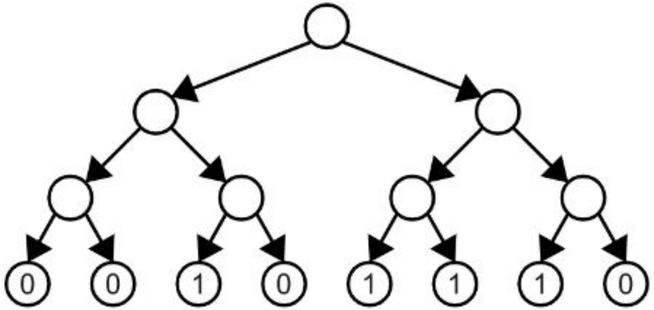
$$\le O\left[n^{\varepsilon'}\right] \qquad \boxed{\varepsilon' = \varepsilon - \frac{(\varepsilon+1)}{2k} < 1}$$

$$\Rightarrow \boxed{T_0(n) = O(n^{\varepsilon})}$$

Convergence of probabilities of random NAND trees!

What is a random NAND tree?

- It is a NAND tree such that all its leaves have randomly assigned values. (i.e 0 or 1)
- All other internal nodes are usual NAND gates and evaluated using their children.
- Ultimate goal is to evaluate the NAND tree i.e find the value of the root.



Comparative performance of the three algorithms

k : a number such that #leaf nodes(n) in a random NAND tree = 2^k

```
Enter k: 13

Naive recursive algorithm: 0.00024497 seconds
Left-first algorithm: 0.00011752 seconds
Random-first algorithm: 2.445e-05 seconds
```

```
Enter k: 18

Naive recursive algorithm: 0.00337031 seconds
Left-first algorithm: 0.00336206 seconds
Random-first algorithm: 9.652e-05 seconds
```

"Fastness" Ratio:

(naive : left_first : random_first)

1 : 2.08 : 100

1 : 1.0024 : 35

Convergence of probabilities of random NAND trees

Notation:

- k : a number such that #leaf nodes(n) = 2^k
- $P_0(k)$: The probability that a random nand tree of 2^k leaf nodes evaluates to 0
- $P_1(k)$: The probability that a random nand tree of 2^k leaf nodes evaluates to 1

The obvious relation:

$P_0(k) + P_1(k) = 1$ (since a NAND tree can evaluate to either 0 or 1)

Convergence of probabilities of random NAND trees

Upshot of the third part of the problem statement:

- We shall prove that $P_0(k)$ and $P_1(k)$ converges to a particular value depending on whether k is odd or even.

Consequence:

- Since probabilities converge for values of k >15, given a random NAND tree of 2^k leaf nodes is formed, it could be virtually told what value the tree would evaluate to (a.k.a with high probability), just by finding k%2.

- CONSTANT TIME!

Mathematical formulations of probability

Notation recap:

- $P_0(k)$: The probability that a random nand tree of 2^k leaf nodes evaluates to 0
- $P_1(k)$: The probability that a random nand tree of 2^k leaf nodes evaluates to 1

For starters:
k = 0 => essentially, 1 node forms the random tree

- $P_0(0) = P_1(0) = 0.5$, since the tree has one node only, it can either take value 0 or 1. Thus, individually probability is 0.5 each

Mathematical formulations of probability

$P_0(k)$	$P_1(k)$
For the root to evaluate to 0, **both** its subtrees **must** evaluate to 1	For the root to evaluate to 1, **atleast** one of its subtrees **must** evaluate to 0
Each subtree contains 2^{k-1} leaf nodes.	Each subtree contains 2^{k-1} leaf nodes.
Thus, $P_0(k) = P_1(k-1) * P_1(k-1)$	We use: $P_0(k) + P_1(k) = 1$
Since, $P_0(k) + P_1(k) = 1$	$\Rightarrow P_1(k) = 1 - P_0(k)$
$\Rightarrow P_0(k) = (1 - P_0(k-1))^2$	$\Rightarrow P_1(k) = 1 - (P_1(k-1))^2$

Convergence of probabilities of random NAND trees

- We have developed recursively formulation for $P_0(k)$ and $P_1(k)$

$$P_0(k) = (1 - P_0(k-1))^2$$

Pseudocode:

```
P_0 (k) {
    if (k = 0)
        return 0.5
    val = P_0(k-1)
    return ((1 - val)*(1-val))
}
```

$$P_1(k) = 1 - (P_1(k-1))^2$$

Pseudocode:

```
P_1 (k) {
    if (k = 0)
        return 0.5
    val = P_1(k-1)
    return (1 - val*val)
}
```

Convergence of probabilities of random NAND trees

```
P0(0): 0.5
P1(0): 0.5

P0(1): 0.25
P1(1): 0.75

P0(2): 0.5625
P1(2): 0.4375

P0(3): 0.191406
P1(3): 0.808594

P0(4): 0.653824
P1(4): 0.346176

P0(5): 0.119838
P1(5): 0.880162

P0(6): 0.774685
P1(6): 0.225315

P0(7): 0.0507667
P1(7): 0.949233

P0(8): 0.901044
P1(8): 0.0989562

P0(9): 0.00979233
P1(9): 0.990208

P0(10): 0.980511
P1(10): 0.0194888

P0(11): 0.000379812
P1(11): 0.99962

P0(12): 0.999241
P1(12): 0.00075948

P0(13): 5.76809e-07
P1(13): 0.999999

P0(14): 0.999999
P1(14): 1.15362e-06

P0(15): 1.33082e-12
P1(15): 1

P0(16): 1
P1(16): 2.66165e-12

P0(17): 0
P1(17): 1

P0(18): 1
P1(18): 0

P0(19): 0
P1(19): 1

P0(20): 1
P1(20): 0
```

We observe that:

For **odd k**: $P_0(k)$ converges to 0
This means, $P_1(k)$ converges to 1

For **even k**: $P_0(k)$ converges to 1
This means, $P_1(k)$ converges to 0

Convergence is obtained roughly for k more than or equal to 15

Consequence of convergence of probabilities

For **odd k**: $P_0(k)$ converges to 0. This means, $P_1(k)$ converges to 1

For **even k**: $P_0(k)$ converges to 1. This means, $P_1(k)$ converges to 0

This means, given **k** (not very small), i.e a random NAND tree with 2^k leaf nodes:

compute (k%2)
if k is odd: this means $P_1(k) = 1$, thus the random tree evaluates to 1!
else if k is even: this means $P_0(k) = 0$, thus the random tree evaluates to 0!

Consequence of convergence of probabilities

- By just finding whether k is odd or even, we have evaluated the random NAND tree (with high probability) in **constant time**
- This is done without actually evaluating the random NAND tree
- Comparatively, if we had to evaluate the random NAND tree using random_first, it would take us sublinear time in expectation.
- So, using probabilistic analysis, we have done way better in evaluating the random NAND tree!

Code up on:

https://github.com/adityabasu1/Evaluating-NAND-Trees

Thank you!