On the Construction of Universal Series-Parallel Functions for Logic Module Design

F.Y. Young and D.F. Wong
Department of Computer Sciences
The University of Texas at Austin
fyyoung@cs.utexas.edu and wong@cs.utexas.edu

Abstract

The structural tree-based mapping algorithm is an efficient and popular technique for technology mapping. In order to make good use of this mapping technique, it is desirable to design logic modules based on Boolean functions which can be represented by a tree of gates (i.e. series-parallel or SP functions). In FPGA-96, Thakur and Wong [5] studied this issue and demonstrated the advantages of designing logic modules as universal SP functions, i.e. SP functions which can implement all SP functions with a certain number of inputs. However, the universal SP functions presented in [5] were designed manually and an automatic generation of universal SP functions was still left as an open problem. In this report, we present an algorithm to generate, for each $n > 0$, a universal SP function for implementing all n-input SP functions. We will also present an efficient Boolean matching algorithm for matching functions to the universal SP functions that we constructed. As it is important to have alternative universal SP functions from which logic-module designers can choose a design taking other criteria (e.g. area, delay, or power) into consideration, we developed an algorithm to generate alternative universal SP functions. In particular, we have found characterized all the universal SP functions for n-input SP functions, when $n \leq 6$.

1 Introduction

Designing a logic module that can implement many different functions is a good idea only if one can come up with a mapping algorithm that can utilize the functionality. Recently, high-functionality logic modules based on universal logic modules (ULMs) have been reported [1, 2, 3, 6, 4], but current technology mappers cannot exploit all the functionality offerred. Typically, the best mapping algorithm for logic modules are discovered after the architecture design has been done. Exploration of logic module architectures revolves around established designs with incremental modifications. In FPGA-96, Thakur and Wong [5] took a dual approach; they began with a known mapping algorithm and designed logic modules for which the mapping algorithm can perform well. The structural tree-based mapping algorithm is an efficient and popular technique for technology mapping. Due to the decomposition of unmapped logic networks into trees, matches will be identified only for those library cells that have a representation in the form of a tree of gates. The mapping algorithm is optimal for libraries restricted to such functions (which will be referred to as series-parallel or SP functions). For a FPGA logic module, the library is the set of functions that can be implemented using one logic module. In order to make good use of this mapping technique, Thakur and Wong [5] designed logic modules as universal SP functions, i.e. SP functions which can implement all SP functions with a certain number of inputs. However, the universal SP functions presented in [5] were designed manually and an automatic generation of universal SP functions was still left as an open problem.

In this report, we present an algorithm to generate, for each $n > 0$, a universal SP function for implementing all n-input SP functions. We will also present an efficient boolean matching algorithm for matching functions to the universal SP functions we constructed. As it is important to have alternative universal SP functions from which logic-module designers can choose a design taking other criteria (e.g. area, delay, or power) into consideration, we developed an algorithm to generate alternative universal SP functions with minimum number of inputs. In particular, we have found all the universal SP functions for n-input SP functions, when $n \leq 6$.

The rest of the report is organized as follows. In Section 2, we formally introduce the problem of constructing universal SP functions with minimum number of inputs and show its equivalence to the problem of finding universal trees with minimum number of leaves. We present an algorithm to construct universal trees (and hence universal SP functions) in Section 3 and give the boolean matching algorithm in Section 4. Finally, in Section 5, we present an algorithm to generate alternative universal SP functions and show all universal SP functions for n-input SP functions, when $n \leq 6$.

2 Formulation of Problem

We denote the complement of a boolean function f as f'. We now formally define *series-parallel (SP) functions* and the notion of a function f *implements* another function g as

follows:

Definition 1 Any function of at most one input is an SP function. If f and g are two SP functions with disjoint supports, then $f + g$ and $f * g$ are SP functions.

Definition 2 For any $m \geq n$, we say that a function $f(z_1, z_2, \cdots, z_m)$ can implement a function $g(x_1, x_2, \cdots, x_n)$ if f can be transformed to g by: (i) Assigning a value from $\{0, 1, x_1, x_1', \cdots, x_n, x_n'\}$ to each of the $z_1, z_2, \cdots, z_m$; and (ii) optionally complementing the output of f.

Example 1 Let $f = ((x_1 + x_2)x_3' + x_4')x_5x_6$ and $g_1 = y_1y_2y_3 + y_4$. Both f and g_1 are SP functions. Putting $x_1 = y_1'$, $x_2 = y_2'$, $x_3 = 0$, $x_4 = y_3$, $x_5 = y_4'$ and $x_6 = 1$ and taking the complement of the output in f gives $(((y_1' + y_2')1 + y_3')y_4' \cdot 1)' = y_1y_2y_3 + y_4 = g_1$. Therefore, f can implement g_1.

Definition 3 Two functions f and g are NPN-equivalent if and only if f can be transformed to g by some combination of input permutations, input complementations and output complementation.

We call an SP function *n-universal* if it can implement all SP functions with at most n inputs. The goal of our work is to construct n-universal SP functions for all $n > 0$. It was proved in [5] that if a function f can implement an SP function g, then f can implement every SP function which is NPN-equivalent to g. Therefore, to construct a n-universal SP function, it suffices to construct a function which can implement one function from each NPN-equivalent class.

Example 2 Let $f = ((x_1 + x_2)x_3' + x_4')x_5x_6$, $g_1 = y_1y_2y_3 + y_4$ and $g_2 = (y_2' + y_3 + y_4')y_1'$. Putting $y_1 = y_2$, $y_2 = y_3'$, $y_3 = y_4$ and $y_4 = y_1$ and taking the complement of the output in g_1 gives $(y_2y_3'y_4 + y_1)' = (y_2' + y_3 + y_4')y_1' = g_2$. Therefore, g_1 and g_2 are NPN-equivalent. From Example 1, we know that f can implement g_1. Since g_1 and g_2 are NPN-equivalent, f should also be able to implement g_2. This is true since putting $x_1 = y_2'$, $x_2 = y_3$, $x_3 = 0$, $x_4 = y_4$, $x_5 = y_1'$ and $x_6 = 1$ in f gives $((y_2' + y_3)1 + y_4')y_1' \cdot 1 = (y_2' + y_3 + y_4')y_1' = g_2$.

An SP function with m inputs can be represented by a *labelled tree* with the following properties:

1. The internal nodes are labelled AND (*) or OR (+) and have at least two children each. The node labels alternate between AND and OR on any path from the root to the leaves.

2. The tree has m leaf nodes. Each leaf node is labelled by one of $\{x_1, x_1', \cdots, x_m, x_m'\}$ such that each variable appears exactly once in some phase.

The *unlabelled tree* of an SP function f is the tree obtained by removing the node labels. Figure 1 shows an SP function and its labelled and unlabelled tree representations. Clearly

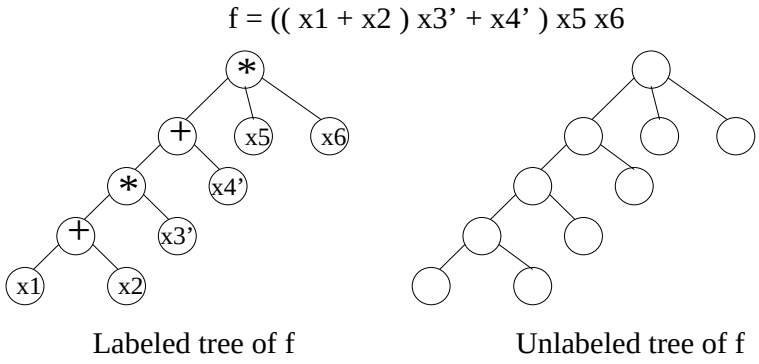


Figure 1: An SP function and its labelled and unlabelled tree representations

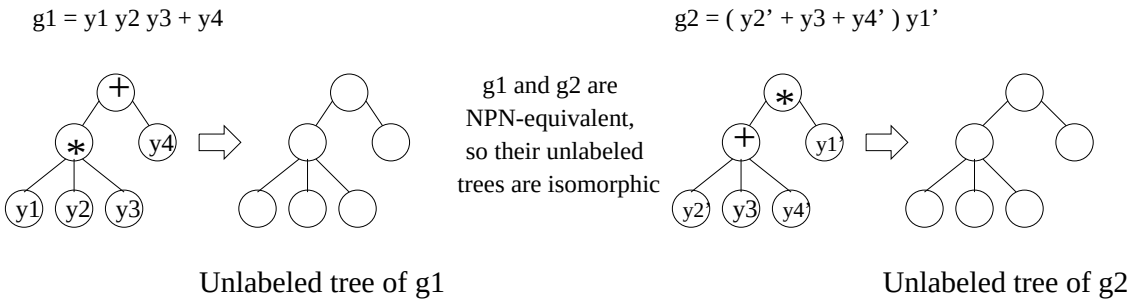


Figure 2: Isomorphism between the unlabelled trees of two NPN-equivalent SP functions

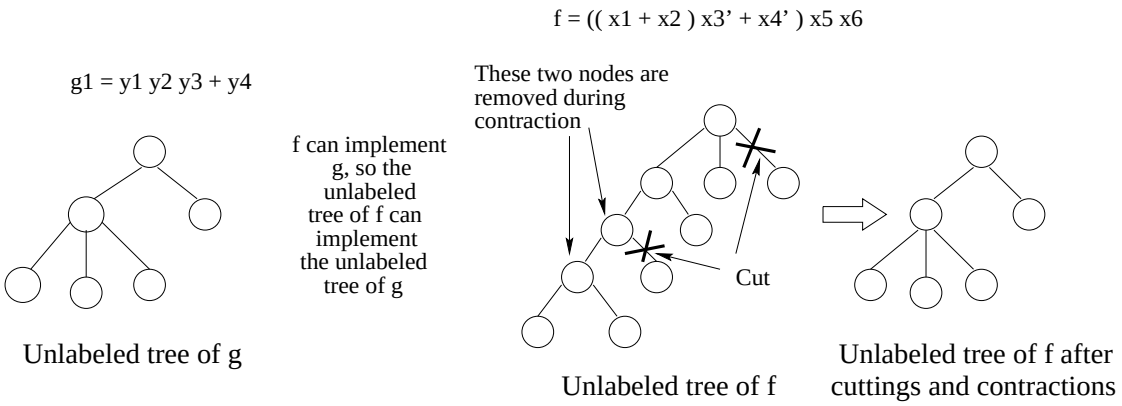


Figure 3: Illustration of the cutting and contraction operations

any SP function yields a unique unlabelled tree (up to isomorphism), but an unlabelled tree may correspond to many different SP functions (by labeling the nodes differently). It was proved in [5] that two SP functions are NPN-equivalent if and only if their unlabelled trees are identical up to isomorphism. This is illustrated by an example in Figure 2. It follows immediately that there is a one-to-one correspondence between the unlabelled trees with m leaves and the NPN-equivalent classes of all m-input SP functions. We now define two operations, cutting and contraction, on unlabelled trees:

Cutting: Two nodes a and b, such that a is a child of b, are selected. The entire subtree rooted at a and the edge between a and b are removed.

Contraction: An internal node b, which has parent a and a single child c, is selected. Node b is removed. If c is an internal node, the children of c are made children of a and c is removed. If c is a leaf, it becomes a child of a.

Let t and t' be two unlabelled trees. We say that t implements t' if t' can be obtained by applying a sequence of cutting or contraction operations to t. Let f and g be two SP functions and let t and t' be their respective unlabelled trees. It was proved in [5] that f implements g if and only if t implements t'. (For example, in Figure 3, we have f implements $g1$ and the unlabelled tree of f can also implement the unlabelled tree of $g1$.) As a result, the following two problems are equivalent.

Logic Module Design Problem: Given an integer $n > 0$, find an SP function f with the minimum number of inputs which can implement all SP functions with at most n inputs.

Universal Tree Design Problem: Given an integer $n > 0$, construct an unlabelled tree T_n with the minimum number of leaf nodes which can implement all unlabelled tree with at most n leaves.

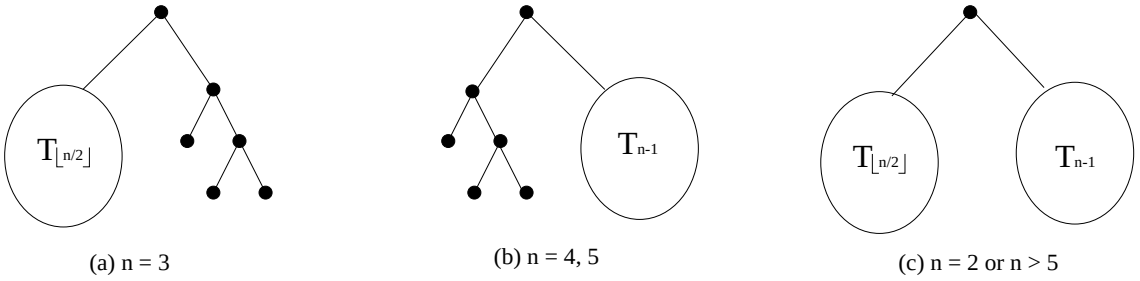


Figure 4: Construction of universal trees

Since the above two problems are equivalent, we will work on the second one from now onwards. Unless otherwise stated, all trees in the following are unlabelled. A tree is *n-universal* if it can implement all unlabelled trees with at most n leaf nodes. The *size* of a tree is defined as the number of leaf nodes.

3 Constructing Universal Trees

3.1 Construction

The universal tree T_n for n-leaf trees is constructed recursively from $T_{\lfloor \frac{n}{2} \rfloor}$ and T_{n-1}:

Algorithm U-TREE: Construct a tree T_n which is n-universal.
Input: A positive integer n.
Output: A tree T_n which is n-universal.
Construction:

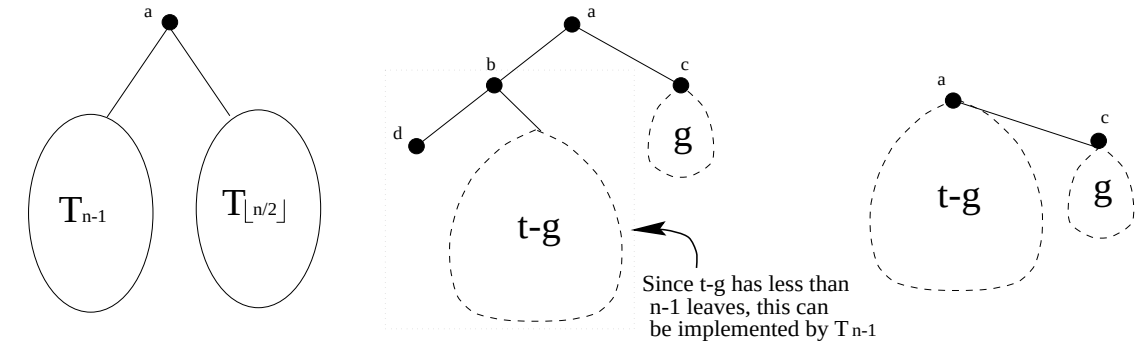
1. If $n = 1$, construct T_n as a single node tree.
2. Else if $n = 3$, construct T_n as in Figure 4(a).
3. Else if $n = 4, 5$, construct T_n as in Figure 4(b).
4. Otherwise, construct T_n as in Figure 4(c).

3.2 Proof of Correctness

It is obvious that T_1 and T_2 are correct, so we consider $n > 2$ only. Let t be a tree with n leaves. Let $t_1, t_2, ..., t_j$ be the subtrees at the root of t where $j \geq 2$. We consider the following three cases:

Case 1: One subtree has only one leaf while the other has $n - 1$ leaves.

It is obvious that T_n can implement t in this case.



(a) Use T_{n-1} to implement t-g (b) Cut(b,d) and Contract(b) (c) t is obtained

Figure 5: Use T_{n-1} to implement $t - g$

Case 2: There exists one subtree g of m leaves where $2 \leq m \leq \lfloor \frac{n}{2} \rfloor$.

This case does not apply to $n = 3$ since $\lfloor \frac{n}{2} \rfloor < 2$ when $n = 3$. Let $t - g$ denotes the set of subtrees at the root of t save g. Since g has more than one leaf, $t - g$ has less than $n - 1$ leaves. Thus we can implement $t - g$ by T_{n-1} (Figure 5), and implement g by $T_{\lfloor \frac{n}{2} \rfloor}$ (or by the left subtree at the root of the tree shown in Figure 4(b) when $n = 4, 5$).

Case 3: Otherwise.

There must be at least two single-leaf subtrees, g_1 and g_2, at the root of t. When $n = 3$, it is the case when the root has three single-leaf children and it is obvious that T_3 can implement this. Consider the case when $n > 3$. Let $t - g_1 - g_2$ denotes the set of subtrees at the root of t except g_1 and g_2. Since g_1 and g_2 has one leaf each, $t - g_1 - g_2$ has less than $n - 1$ leaves. Hence we can implement $t - g_1 - g_2$ by T_{n-1}, and implement g_1 and g_2 by $T_{\lfloor \frac{n}{2} \rfloor}$ (or the left subtree at the root of the tree shown in Figure 4(b) when $n = 4, 5$).

3.3 Analysis

Let $f(n)$ be the size of the universal tree T_n constructed by U-TREE. We will show that $f(n)$ lies between $n^{\frac{\lg n}{4}}$ and $n^{\frac{\lg n+1}{2}}$ when n is large. From the construction, we know that $f(1) = 1$, $f(2) = 2$, $f(3) = 4$, $f(4) = 7$, $f(5) = 10$ and

$$f(n) = f(\lfloor \tfrac{n}{2} \rfloor) + f(n-1) \qquad \text{when } n > 5$$

So, for $n > 5$,

$$f(n) = f(5) + 2\sum_{i=3}^{\lfloor \frac{n}{2} \rfloor} f(i) \qquad \text{when } n \text{ is odd}$$

$$f(n) = f(5) + f(\lfloor \tfrac{n}{2} \rfloor) + 2\sum_{i=3}^{\lfloor \frac{n}{2} \rfloor - 1} f(i) \qquad \text{when } n \text{ is even}$$

Therefore

$$2\sum_{i=1}^{\lfloor \frac{n}{2} \rfloor - 1} f(i) \leq f(n) \leq 4 + 2\sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} f(i) \qquad \text{for all } n$$

For simplicity, lets assume that n is a positive power of 2. Then

$$f(n) \geq 2 \times \tfrac{n}{4} \times f(\tfrac{n}{4}) = \tfrac{n}{2} \times f(\tfrac{n}{4}) \geq \tfrac{n}{2} \times \tfrac{n}{8} \times f(\tfrac{n}{16}) \geq \cdots \geq n^{\frac{\lg n}{4}}$$
$$f(n) \leq 2 \times \tfrac{n}{2} \times f(\tfrac{n}{2}) = n \times f(\tfrac{n}{2}) \leq n \times \tfrac{n}{2} \times f(\tfrac{n}{4}) \leq \cdots \leq n^{\frac{\lg n+1}{2}}$$

Therefore

$$n^{\frac{\lg n}{4}} \leq f(n) \leq n^{\frac{\lg n+1}{2}}$$

3.4 Comparison with Lower Bounds

We have proved the following theorem which gives a lower bound on the size of a universal tree:

Theorem 1 *The size of a n-universal tree is at least*

$$\sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \lfloor \frac{n}{i} \rfloor + \sum_{i=1}^{\lfloor \frac{n-1}{2} \rfloor} \lfloor \frac{n-1}{i} \rfloor - \lfloor \frac{n}{2} \rfloor - \lfloor \frac{n-1}{2} \rfloor + 1$$

Table 1 compares the size of the universal trees constructed by U-TREE with the lower bound from Theorem 1. We can see that U-TREE gives optimal universal trees up to n equals 7 and the solution is off from optimal by at most 5 when n is less than 10.

4 Boolean Matching

In boolean matching, we are given two boolean functions f and g. We want to know whether function g can implement function f and to construct f from g in case it is possible. In this section, we describe a polynomial time algorithm which, when given a boolean function f expressed as a tree t and a universal tree T_n constructed by U-TREE, can determine whether T_n can implement t by some sequence of cuttings or contractions in $O(m \log D)$ time where m is the number of leaves in t and D is the largest fan-in in t. This problem is non-trivial when $m > n$. The construction will also be found if the answer is yes. In this section, we only consider the universal trees T_k'sconstructed by U-TREE. We need the following two claims in the algorithm.

n	Size from U-TREE	Lower Bound
1	1	1
2	2	2
3	4	4
4	7	7
5	10	10
6	14	14
7	18	18
8	25	22
9	32	27

Table 1: Comparison between U-TREE and the lower bound

Lemma 1 *If $i > j$, T_i can implement T_j.*

Proof The proof is done by induction. It is easy to verify that T_k can implement T_{k-1} for $k \leq 6$. Consider the construction of T_k for $k > 6$. By the inductive hypothesis, its left subtree $T_{\lfloor \frac{k}{2} \rfloor}$ can implement $T_{\lfloor \frac{k-1}{2} \rfloor}$ and its right subtree T_{k-1} can implement T_{k-2}. Therefore T_k can implement T_{k-1}.

□

Lemma 2 *Given t with d subtrees at the root. Let $T_{n_1}, T_{n_2}, \cdots, T_{n_d}$ be the smallest universal trees to implement these subtrees. (According to Lemma 1, they are well defined.) If T_z is the smallest universal tree to implement t, then $p \leq z \leq p + 2d$ where $p = \max\{2y, x + 1\}$, x is the largest n_i and y is the second largest n_i.*

Proof Consider a tree t rooted at a node v. Assume that v has d children subtrees, $g_1, g_2, \ldots g_d$ and that g_1 and g_2 are the largest and the second largest children subtrees respectively. Let T_x and T_y be the smallest universal trees to implement g_1 and g_2 respectively. We want to show that $p \leq z \leq p + 2d$ where T_z is the smallest universal tree to implement t and $p = \max\{2y, x + 1\}$. We consider two different cases:

Case 1: $x + 1 \geq 2y$, so $p = x + 1$.

We want to show that $x + 1 \leq z \leq (x + 1) + 2d$.

Consider T_z where $z < x + 1$. The two subtrees at the root are T_{z-1} and $T_{\lfloor \frac{z}{2} \rfloor}$. Since $z < x + 1$, neither T_{z-1} nor $T_{\lfloor \frac{z}{2} \rfloor}$ can implement g_1. Thus $z \geq x + 1$.

Consider T_z where $z = (x + 1) + 2d$. The two subtrees at the root are T_{z-1} and $T_{\lfloor \frac{z}{2} \rfloor}$. If we expand T_{z-1} to T_{z-2} and $T_{\lfloor \frac{z-1}{2} \rfloor}$, cut the subtree $T_{\lfloor \frac{z-1}{2} \rfloor}$ and do contraction, we get T_{z-3}, $T_{\lfloor \frac{z}{2} \rfloor - 1}$ and $T_{\lfloor \frac{z}{2} \rfloor}$ at the root. We can repeat this process d times and obtain $d + 2$ subtrees at the root: T_{z-1-2d}, $T_{\lfloor \frac{z}{2} \rfloor - d}$, $T_{\lfloor \frac{z}{2} \rfloor - d + 1}$, $\cdots$, $T_{\lfloor \frac{z}{2} \rfloor}$. Since $z = (x + 1) + 2d$,

$z-1-2d = x$ and $\lfloor \frac{z}{2} \rfloor - d = \lfloor \frac{x+1}{2} \rfloor \geq y$. Thus the T_{z-1-2d} can implement g_1 and the $T_{\lfloor \frac{z}{2} \rfloor - d}$ can implement g_2. Since g_2 is already the second largest subtree in t, g_3, g_4, $\cdots$, g_d in t can be implemented by the remaining d subtrees in T_z.

Case 2: $2y \geq x+1$, so $p = 2y$.

We want to show that $2y \leq z \leq 2y + 2d$.

Consider T_z where $z < 2y$. The two subtrees at the root are T_{z-1} and $T_{\lfloor \frac{z}{2} \rfloor}$. Since $z < 2y$, $T_{\lfloor \frac{z}{2} \rfloor}$ cannot implement g_2. The only possibility is to expand T_{z-1} giving T_{z-3}, $T_{\lfloor \frac{z}{2} \rfloor - 1}$ and $T_{\lfloor \frac{z}{2} \rfloor}$. However neither $T_{\lfloor \frac{z}{2} \rfloor - 1}$ nor $T_{\lfloor \frac{z}{2} \rfloor}$ can implement g_2 and the only possibility is to expand T_{z-3}. Repeating the same argument, we finally get a subtree just large enough to implement g_1 but none of the others can implement g_2. Thus $z \geq 2y$.

Consider T_z where $z = 2y + 2d$. The two subtrees at the root are T_{z-1} and $T_{\lfloor \frac{z}{2} \rfloor}$. If we expand T_{z-1}, we get T_{z-3}, $T_{\lfloor \frac{z}{2} \rfloor - 1}$ and $T_{\lfloor \frac{z}{2} \rfloor}$ at the root. We can repeat this process d times and obtain $d+2$ subtrees at the root: T_{z-1-2d}, $T_{\lfloor \frac{z}{2} \rfloor - d}$, $T_{\lfloor \frac{z}{2} \rfloor - d + 1}$, $\cdots$, $T_{\lfloor \frac{z}{2} \rfloor}$. Since $z = 2y + 2d$, $z - 1 - 2d = 2y - 1 \geq x$ and $\lfloor \frac{z}{2} \rfloor - d = y$. Thus the T_{z-1-2d} can implement g_1 and the $T_{\lfloor \frac{z}{2} \rfloor - d}$ can implement g_2. Since g_2 is already the second largest subtree in t, g_3, g_4, $\cdots$, g_d in t can be implemented by the remaining d subtrees in T_z.

□

4.1 Boolean Matching Algorithm

According to Lemma 1, we can determine whether T_n can implement an m-leaf tree t by finding the smallest index k such that T_k can implement t. The algorithm works bottom-up from the leaves to the root. Assuming that we know already the smallest universal trees to implement the d_v children subtrees of a node v, we can do binary search on $T_p, \cdots, T_{p+2d_v}$ to find the smallest universal tree to implement the tree rooted at v, where, according to Lemma 2, p is $\max\{2y, x+1\}$, x is the largest index among the universal trees for v's children subtrees and y the second largest. This can be done by the following algorithm MATCH. We can apply this algorithm recursively from the leaves up to the root until we find the smallest T_k to implement the entire tree t at the root. In the following, *decomposing* a universal tree means the sequence of steps shown in Figure 6 to obtain smaller universal trees at the same level.

Algorithm MATCH
Input: A tree t with d subtrees $t_1, t_2, \ldots, t_d$ at the root such that $T_{n_1}, T_{n_2}, \ldots, T_{n_d}$ are the smallest universal trees to implement these subtrees. An integer n.
Output: Check if T_m can implement t. If yes, give construction.

1. Let $A = \{n_1, n_2, \ldots, n_d\}$.

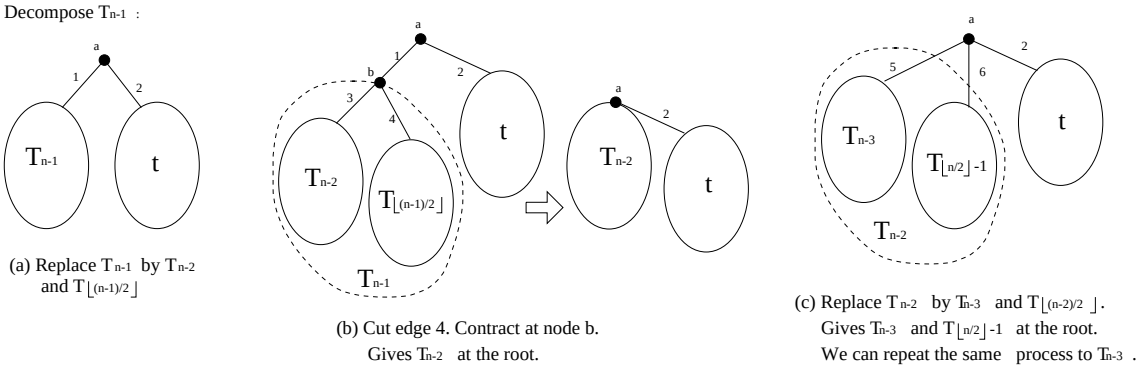


Figure 6: Decomposing a universal tree

2. Let $B = \{m-1, \lfloor \frac{m}{2} \rfloor\}$.
3. While A is not empty
   (a) Let i be the largest index in A, corresponding to subtree t_i at the root of t.
   (b) Let j be the largest index in B with the same parity of i.
   (c) Let k be the largest index in B with the opposite parity of i.
   (d) If $(j < i)$ and $(k < i)$, exit and output FAIL
   (e) Else if $(j \geq i)$:
      i. Decompose T_j into smaller universal trees until getting T_i. Use this T_i to implement t_i.
      ii. Let B' be the set of indices of the universal trees obtained by decomposing T_j.
      iii. $B = B - \{j\} + B' - \{i\}$
      iv. Output the construction.
   (f) Else if $(k \geq i)$ and $(\lfloor \frac{k}{2} \rfloor < i)$:
      i. Decompose T_k into smaller universal trees until getting T_{i+1}. Use this T_{i+1} to implement t_i.
      ii. Let B' be the set of indices of the universal trees obtained by decomposing T_k.
      iii. $B = B - \{k\} + B' - \{i+1\}$
      iv. Output the construction.
   (g) Else if $(k \geq i)$ and $(\lfloor \frac{k}{2} \rfloor \geq i)$:
      i. Decompose T_k into smaller universal trees until getting T_i. Use this T_i to implement t_i.
      ii. Let B' be the set of indices of the universal trees obtained by decomposing T_k.
      iii. $B = B - \{k\} + B' - \{i\}$
      iv. Output the construction.

(h) $A = A - \{i\}$

(i) End {while}

4. End {MATCH}

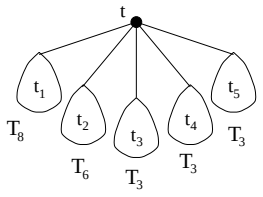
An example is shown in Figure 7. In the algorithm, A keeps a set of indices of the subtrees which we need to implement while B is a set of indices of the available subtrees. The while-loop examines the subtrees corresponding to the indices in A in a non-increasing order. In each iteration, the largest index i in A is picked and we want to check whether there is an available subtree whose index is at least i. Thus we pick the largest odd index and the largest even index from B. If both are smaller than i, we know that t_i cannot be implemented by any available subtree. Otherwise, we prefer implementing t_i by T_j, whose index j has the same parity as i, because decomposing T_j will give T_i exactly. However if $j < i$, we must use T_k, whose index k has the opposite parity as i. If $\lfloor \frac{k}{2} \rfloor \geq i$ we can still implement t_i exactly by decomposing T_k to give T_i. Otherwise we must waste some resources by implementing t_i by T_{i+1} obtained by decomposing T_k. Notice that T_k can never give T_i unless $\lfloor \frac{k}{2} \rfloor \geq i$. After that, we need to update A and B by removing i from A, removing j (or k) from B and adding back $B' - \{i\}$ (or $B' - \{i+1\}$) to B where B' is the set of indices obtained by decomposing T_j (or T_k). This process repeats until either there is a subtree in A which cannot be implemented or A is empty. Since we examine the indices in A in decreasing order, we will not mistakenly decompose a universal tree in B which is needed in some later steps. Thus the algorithm is correct. As a minor implementation detail, we need some special data structure for the sets A and B such that the largest element can be picked quickly in each step.

Let $TIME(v, j)$ denotes the time to check whether a universal tree T_j can implement a tree rooted at v, given the smallest universal trees to implement v's children subtrees. From the above algorithm, we know that $TIME(v, j) = O(d_v)$ where d_v is the in-degree of v. Thus the total time spent at node v will be $\log d_v \times O(d_v)$ where the logarithmic term comes from the binary search. Let D be the largest fan-in in t, then

$$TOTAL\ TIME\ = \sum_v O(d_v \log d_v) = O(\log D^{\sum_v d_v}) = O(m \log D)$$

5 Alternative Universal Trees

During the design process, it is advantageous to have alternative universal trees so that the designers have options to choose from. The final choice may be based on functionality, area, delay, power or any other considerations. When n is small ($1 \leq n \leq 6$), we can generate all optimal (smallest number of leaves) n-universal trees Since algorithm 1 constructs universal trees recursively from smaller ones, we can generate many alternatives for large n from those alternatives for small n.

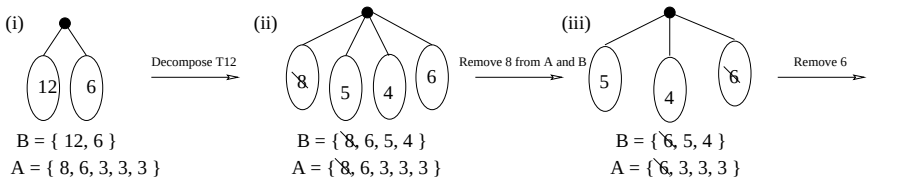


The original tree

x = 8
y = 6
p = max { x+1, 2y } = 12

Thus $12 \leqslant z \leqslant 12 + 2d = 22$
where T_z is the smallest universal tree to implement t

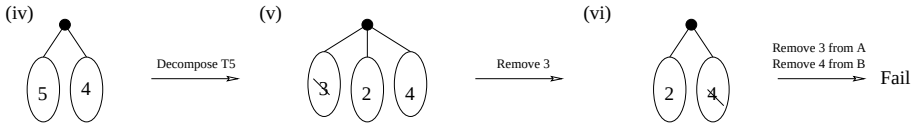
(a) Check if z = 13



B = { 12, 6 }
A = { 8, 6, 3, 3, 3 }

B = { 8, 6, 5, 4 }
A = { 8, 6, 3, 3, 3 }

B = { 6, 5, 4 }
A = { 6, 3, 3, 3 }

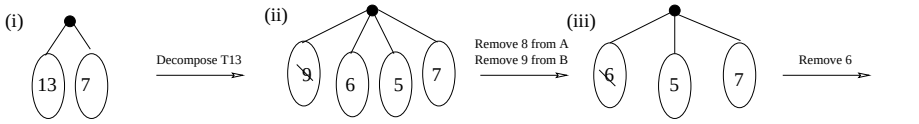


B = { 5, 4 }
A = { 3, 3, 3 }

B = { 4, 3, 2 }
A = { 3, 3, 3 }

B = { 4, 2 }
A = { 3, 3 }

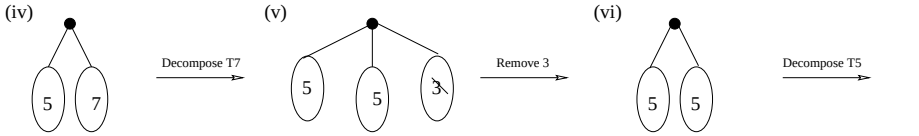
(b) Check if z = 14



B = { 13, 7 }
A = { 8, 6, 3, 3, 3 }

B = { 9, 7, 6, 5 }
A = { 8, 6, 3, 3, 3 }

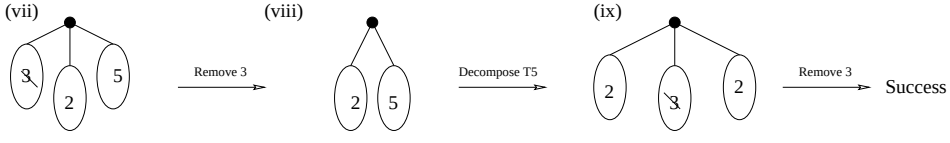
B = { 7, 6, 5 }
A = { 6, 3, 3, 3 }



B = { 7, 5 }
A = { 3, 3, 3 }

B = { 5, 5, 3 }
A = { 3, 3, 3 }

B = { 5, 5 }
A = { 3, 3 }



B = { 5, 3, 2 }
A = { 3, 3 }

B = { 5, 2 }
A = { 3 }

B = { 3, 2, 2 }
A = { 3 }

Figure 7: Running the algorithm MATCH on an example

The problem of generating all optimal universal trees is non-trivial. The running time is unbearably long even when n is small. For example, a universal tree for 6-leaf trees has at least 14 leaf nodes and there are 8005 of them. We need to check, for each one, whether it can generate all possible 6-leaf trees. In order to get the results in a reasonable amount of time, we need a good representation of the trees and an efficient strategy to eliminate non-universal trees. In the following, we will describe the data structures we used and give a brief outline of the algorithm.

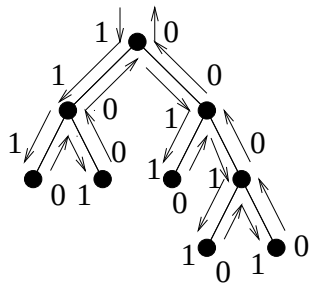
5.1 Data Structures

We use 0-1 bit strings to represent trees. Each node is represented by a pair of 0 and 1. To obtain a binary representation of a tree t, we traverse t in a depth-first order. We write a 1 when we first visit a node and write a 0 when we backtrack from it (see Figure 8(a)). This gives a compact representation for the trees, and, more importantly, it allows us to make use of some fast bitwise operations (e.g. AND, OR, SHIFT) in the C programming language. The drawback of this representation is that the maximum size of the trees on which we can operate is bounded by the wordsize of the computer used. However a 64-bit (type long long in C) wordsize is already enough to generate universal trees for practical purposes and we can also simulate the bitwise operations on an array of words for large trees. The following two procedures are examples on how we can operate on the trees efficiently using this kind of representation:

```
————————————————————
Cut a subtree in tree t and count
the number of edges in the
removed subtree.
————————————————————
int CutSubtree(t)
unsigned long *t;
{   int count=1, edge=0;
        while (count>0) {
        *t>>=1;
        if (*t&0x00000001) count—;
        else count++, edge++;
    }
    return edge;
}
```

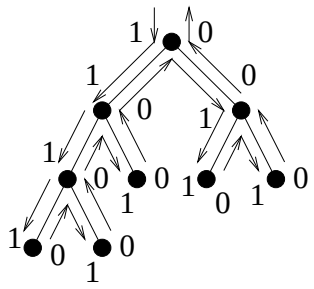
```
————————————————————
Find all the single-child internal
nodes in tree t and mark them as
"0" in the flag r
————————————————————
int FindContractableEdge(t,r)
unsigned long *t, *r;
{   int count=0, type=0;
    if (*t&0x00000001) return 0;
    *t>>=1;
    while (!(*t&0x00000001)) {
        type = FindContractableEdge(t,r);
        count += (type==2)?type:1;
    }
    *t>>=1;
    if (count==0) {*r=(*r<<=1)|0x00000001; return 0;}
    if (count> 1) {*r=(*r<<=1)|0x00000001; return 1;}
    if (count==1) {*r=(type==0)?
            ((*r<<=1)&0xfffffffe):((*r<<=1)&0xfffffffd);
            return (type==0)?0:2;}
}
```

In order to give unique representations (up to isomorphism), we require that the children subtrees at any node v must be arranged in a non-increasing order of their binary representations (Figure 8(b)). In this way, every tree t will correspond to one single binary



111010011011010000

(a) A binary representation of the tree



111101001001101000

(b) A unique representation by arranging the subtrees in a non-increasing order of their binary representations

Figure 8: Example of a unique binary representation of a tree

representation only. This makes the process of removing duplicate trees easy. To remove duplicates, we can treat the binary strings as integers, sort them and scan the list to remove adjacent duplicate strings.

5.2 Algorithm Outline

The following algorithm generates all optimal n-universal trees given an arbitrary integer n.

Algorithm ALL-U-TREE
Input: An integer n
Output: Generate all n-universal trees with the smallest number of leaves.

1. Generate a list of all n-leaf trees in L_1.
2. Let m be the size of the smallest possible n-universal trees. Generate all m-leaf trees in L_2.
3. $L = \{\}$.
4. For each tree t in L_2:
   (a) Count the number of nodes at the odd levels and the even levels of t to see if it can possibly be an n-universal tree. If it is impossible, go back to the beginning of this loop and check the next tree in L_2.
   (b) $L_3 = \{\}$.
   (c) Consider all possible combinations of *cutting* and *not cutting* at each edge of t:
      i. Do cutting in t.
      ii. Do contraction in t.

iii. Rewrite the binary representation of t in such way that subtrees are arranged in a non-increasing order of their binary representations.

iv. Put t in L_3.

(d) Remove duplicates in L_3.

(e) Compare L_1 with L_3. If $L_1 = L_3$, put t into L.

5. Output L.

There are several places in the algorithm where it is needed to generate all k-leaf trees. To do this, we start with a list of all possible 0-1 bit strings of length $4k-2$ (a k-leaf tree has at most $2k-1$ nodes), and we select from this list those valid ones by checking whether:

- The number of leaves ("10" strings) is k.
- The 1's and 0's are balanced.
- All nodes have at least two children.
- Children subtrees are arranged in a non-increasing order of their binary representations.

In case of generating universal trees, we can have one more checking which counts the number of nodes at the odd levels and the number of nodes at the even levels (root is at level 0) because of the following theorem:

Theorem 2 *A n-universal tree must have at least $n + \sum_{i=2}^{\lfloor \frac{n-1}{2} \rfloor} \lfloor \frac{n-1}{i} \rfloor$ nodes at the odd levels and at least $n - 1 + \sum_{i=2}^{\lfloor \frac{n}{2} \rfloor} \lfloor \frac{n}{i} \rfloor$ nodes at the even levels.*

The proof is very similar to that of Theorem 1 and the proof will be given in another piece of work. When k is large, it is impossible to start with a list of all possible 0-1 bit strings of length $4k-2$. For example, when n is 6, according to Theorem 1 and the algorithm U-TREE, we know that an optimal 6-universal tree has 14 leaf nodes. There are 2^{54} possible combinations of 0-1 bit strings of length 54. It is impossible to start with this infinite list to generate universal trees. In real cases, we do prunings at the beginning of the search to throw away a lot of impossible choices. The rules are based on some simple observations of the balancing of the 0's and 1's in the binary representations. In step 4(c), we consider all possible combinations of *cutting* and *not cutting* at the edges of t. This number looks tremendous but it can be reduced significantly if we notice that cutting an edge allows us to neglect all the edges inside the removed subtree.

5.3 Results

Table 2 shows the number of optimal n-universal trees when $1 \leq n \leq 6$. We see that there can be many different choices among the universal trees and they differ in many aspects like number of levels, number of gates, number of wires, number of fan-in of the gates and, the most important of all, the number of functions covered (functionality). Table 3 shows the number of optimal universal trees which can cover a large number of functions for n from one to six. Their structures are shown in Figure 9.

To convert a tree back to a corresponding SP function, we can choose either an AND or OR at the root of the tree, alternate these labels between levels and assign a variable to each leaf node such that each variable appears exactly once in some phase, e.g. a possible SP function corresponding to the first tree in the list of T_4 in Figure 9 is $(((x_1 + x_2')x_3 + x_4')x_5 + x_6x_7')$.

n	No. of Optimal Universal Trees	No. of Functions Implemented
1	1	1
2	1	2
3	2	5
4	12	17-23
5	70	64-107
6	325	349-853

Table 2: Number of optimal universal trees

n	No. of Optimal Universal Trees of High Functionality	No. of Functions Implemented
1	1	1
2	1	1
3	2	5
4	7	above 20
5	5	above 100
6	7	above 800

Table 3: Number of optimal universal trees of high functionality

References

[1] C. Lin, M. Marek-Sadowska, and D. Gatlin. Universal logic gate for FPGA design. *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, 1994.

[2] Y.N. Patt. Optimal and near-optimal universal logic modules with interconnected external terminals. *IEEE Transactions on Computers*, 22(10):903–907, 1973.

[3] F.P. Preparata. On the design of universal boolean functions. *IEEE Transactions on Computers*, 20(4):418–423, 1971.

[4] S. Thakur and D.F. Wong. On designing ULM-based FPGA logic modules. *Proceedings of SCM/SIGDA FPGA-95*, pages 3–9, 1995.

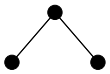
[5] S. Thakur and D.F. Wong. Universal logic modules for series-parallel functions. *Proceedings of ACM/SIGDA FPGA-96*, pages 31–37, 1996.

[6] Z. Zilic and Z.G. Vranesic. Using BDDs to design ULMs for FPGAs. *Proceedings of ACM/SIGDA FPGA-96*, pages 24–30, 1996.

(i) T_1

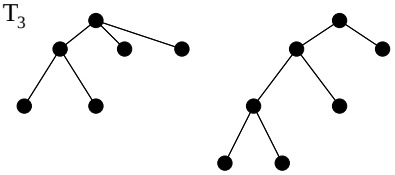
Implements 1 tree

(ii) T_2



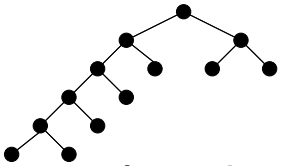
Implements 2 trees

(iii) T_3

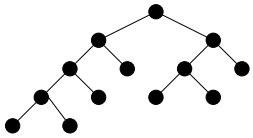


Implements 5 trees

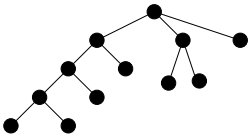
(iv) T_4



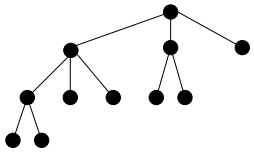
Implements 23 trees



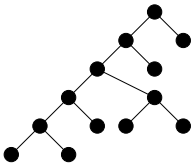
Implements 22 trees



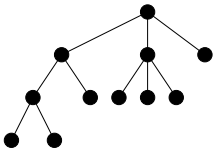
Implements 22 trees



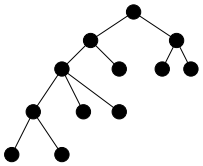
Implements 21 trees



Implements 21 trees

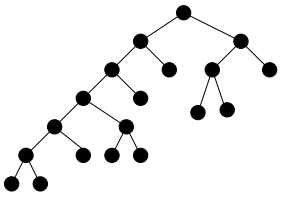


Implements 20 trees

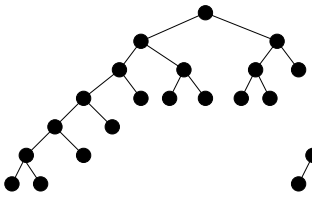


Implements 20 trees

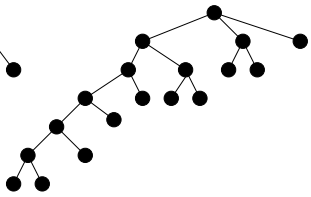
(v) T_5



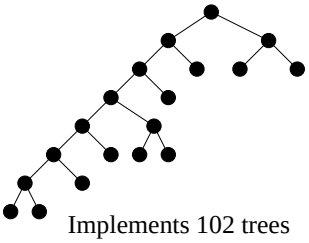
Implements 107 trees



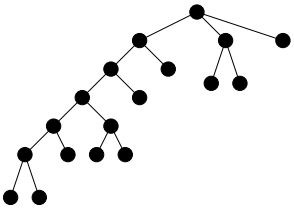
Implements 106 trees



Implements 103 trees



Implements 102 trees



Implements 101 trees

(vi) T_6

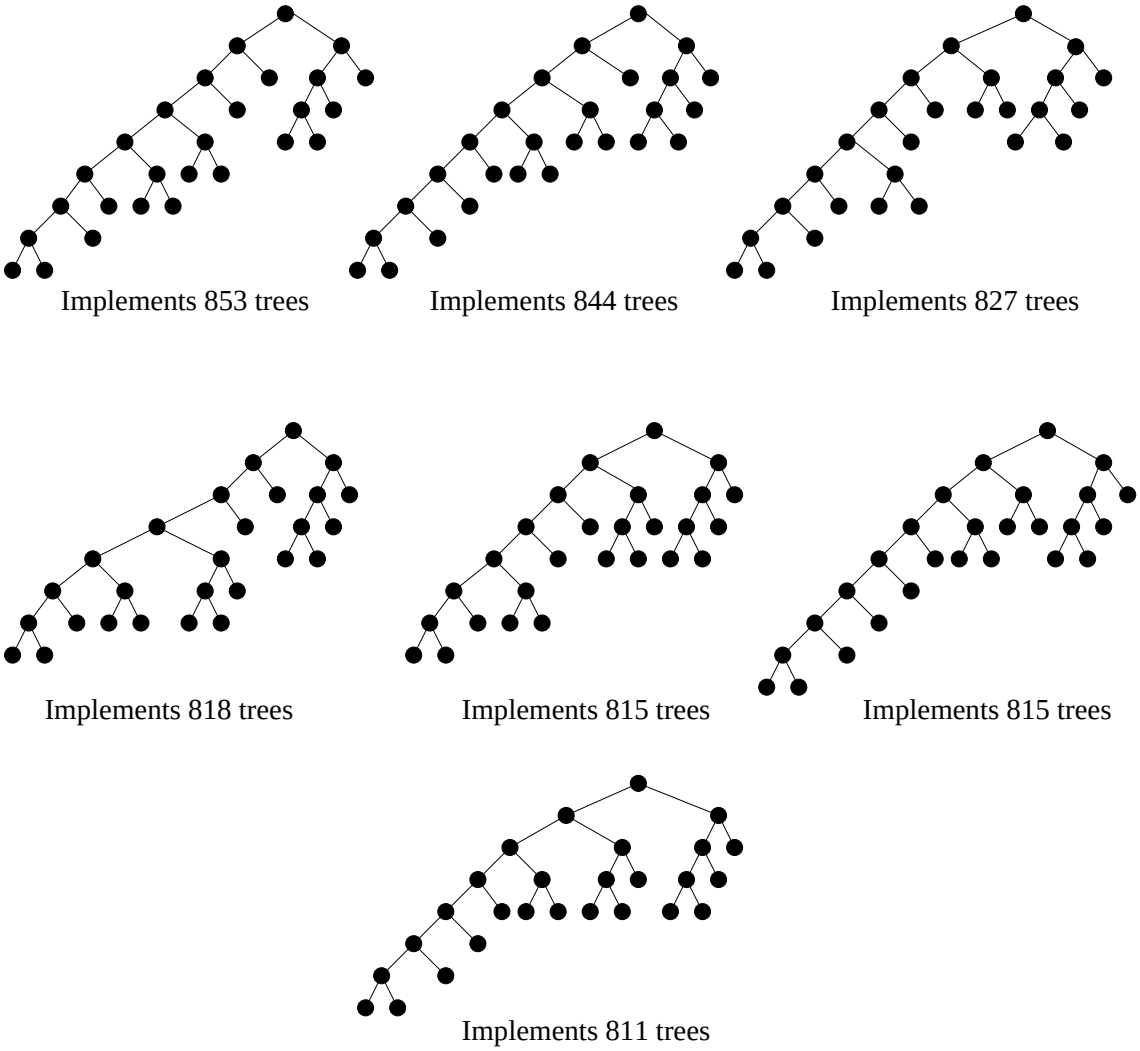


Figure 9: Optimal universal trees of high functionality