Alphabetic Trees: Theory and Applications in Layout-Driven Logic Synthesis

Revision of TCAD Manuscript # 1199

Hirendu Vaishnav
SynApps Software Corp.
1250 Oasis Ct.
Fremont, CA 94539
vaishnav@synappscorp.com

Massoud Pedram
University of Southern California
Department of Electrical Engineering - Systems
Los Angeles, CA 90089-2562
massoud@zugros.usc.edu

Abstract

Routing plays an important role in determining the total circuit area and circuit performance and hence must be addressed as early as possible during the design process. In this paper, an effective routing-driven approach for technology-dependent logic synthesis which relies on alphabetic tree construction is presented. Alphabetic trees are trees which are generated under the restriction that the initial order on the leaf nodes is maintained while not introducing any internal edge crossing.

First, a mechanism for generating all alphabetic trees on a given number of leaf nodes is presented. Next, the number of such trees is calculated under different height and degree restriction and used to derive upper bounds on the complexity of alphabetic tree optimization problem. A classification of tree cost functions for which alphabetic trees can be generated in polynomial time is also proposed. Specifically, alphabetic tree optimization algorithms are applied to generate optimal alphabetic fanout trees. For fanout optimization we obtained 14% improvement in chip area at the cost of 1% loss in performance.

Key words: *Alphabetic trees, Tree enumeration and optimization, Fanout optimization, Routing-driven logic synthesis.*

1 Introduction

With the move towards deep-submicron technology, circuit designers enter a new world in which interconnect becomes a dominant factor in determining all costs associated with VLSI chips. Even in the submicron circuits, routing affects the circuit significantly. Currently, routing accounts for about 40-80% of total chip area, 40-60% of the circuit delay and a significant part of the total power dissipation. A combined effect of ever-increasing dominance of interconnect and synthesis tools that ignore the interconnect effects is that the area, delay, and/or power dissipation constraint violations are increased substantially after the interconnect contribution is taken into account. This results in a dramatic increase in the number of synthesis-layout iterations to meet the area, delay, or power dissipation constraints, increasing the design time significantly. Hence, addressing routing issues at all levels of design abstraction has become a necessity. This paper is a step toward achieving this objective: specifically, it describes a routing-driven approach for the performance-oriented technology dependent phase of logic synthesis using alphabetic trees.

Technology dependent phase of logic synthesis mainly consists of three stages: technology decomposition, technology mapping and fanout optimization. Technology decomposition, which is the procedure for converting an optimized Boolean network into a 2 input NAND-decomposed network, is a precursor to the technology mapping step. Technology mapping problem is the optimization problem of finding a minimum cost covering of this "subject graph" by choosing from a collection of "primitive graphs" constructed for each gate in the library. In past years, technology independent logic synthesis research has concentrated on techniques that maximize logic sharing resulting in circuits with high number of fanouts per net. Excessive loads at such high fanout gates after technology mapping result in considerable performance degradation. Fanout optimization which generates buffer/inverter trees at the output of such heavily loaded gates is thus necessary to improve the circuit performance.

After appropriate technology-independent optimizations, both technology decomposition and fanout optimization essentially become tree optimization problems[1] with different com-

[1]Given a weighted set of leaf nodes, the objective of a tree optimization problem is to generate a tree that optimizes a tree cost function defined on weights of tree nodes calculated using a *combining function* where the combining function is a function using which weight of a parent node is calculated given the weights of its children.

bining and tree cost functions. Hence, algorithms developed for tree optimization are directly applicable to technology decomposition and fanout optimization. In this paper, we develop alphabetic tree generation/optimization algorithms and then apply them to the fanout optimization problem. Appliaction to technology decomposition is straight-forward; It is not included here due to space limitation.

1.1 Prior Work

For fanout optimization, tree optimization algorithm should be able to generate a non-binary tree under a unit fanout delay model or under a library delay model [2].

Golumbic [10] and Hoover et al. [13] addressed fanout optimization under a *unit* delay model with a restriction on maximum number of fanouts per buffer. Golumbic's "combinatorial merging" algorithm [10] is an application of *t-ary* Huffman tree generation procedure to logic synthesis. Hoover et al. [13] present an algorithm to obtain networks of bounded fanin and fanout, so that both size and depth are not increased by more than a constant factor. Unit delay model, however, ignores the load and hence is not adequate for fanout optimization. Berman et al. [1], Singh et al. [35], and Touati [37] used more realistic unit fanout and library delay models. It was shown in [1, 37] that with these delay models, even under very simplistic assumptions, fanout problem is NP-hard. These results have motivated various heuristic solutions. Berman's algorithm generates optimal fanout trees for a restricted set of trees with identical required times for all sinks. Singh's heuristics consists of three operations: repowering, critical signal isolation, and load balancing. Touati's work on fanout optimization is the most comprehensive to date. He extended Golumbic's work to take into account varying loads and variable node degrees for internal nodes of the fanout tree. To integrate *critical signal isolation* with *load balancing*, Touati used "LT-Trees" which balance loads and isolate critical signals simultaneously by grouping signals with similar required times at similar depths of the fanout tree.

[2] Unit delay model assumes that every gate has delay of 1 unit irrespective of the load. *Unit fanout* and *library* delay models are more accurate measures of circuit delay. Under the *unit fanout* delay model, delay of the gate is given by $1 + \alpha \cdot$ *number_of_fanout*, where $0 < \alpha \leq 1$. *Library* delay model uses accurate, pin-dependent values for intrinsic delay and drive of the source as well as accurate load values for the sinks.

1.2 Layout-driven Logic Synthesis using Alphabetic Trees

Using any of the existing fanout optimization mechanisms, if the original circuit graph is planar, the resultant circuit graph may become non-planar. In general, this is not desirable since it tends to increase the routing cost for the resulting circuit graph. Even if the original circuit is non-planar, it is desirable to have a fanout optimization algorithm that does not increase the nonplanarity and hence does not create more routing difficulties.

Alphabetic fanout trees provide a good trade-off between circuit performance and routability. These are the trees that maximize the required time at the root of the fanout tree subject to a fixed linear order on the sinks, without creating any internal edge crossings. Linear order on the output nodes for fanout trees is derived from a "companion placement" solution of the circuit [31]. This placement is incrementally updated during technology mapping and relaxed (to eliminate gate overlaps) after fanout optimization.

The penalty for using alphabetic trees is minimal. It has been shown that under the unit delay model, increase in depth is at most one, and increase in size is a constant multiplicative factor for optimal alphabetic fanout trees as compared to optimal nonalphabetic trees [13, 21].

Instead of using placement information, topological (structural) information to derive the order on the leaf nodes can be used. Topological information is more abstract than the placement information and hence is more appropriate for use whenever the exact gate implementation of the circuit is not known. Other mechanisms to order leaf nodes during fanout optimization can be used as well. For example, an ordering based on required times at the leaf nodes of a fanout tree can be used on the premise that sinks with similar required time are likely to be on the same level of the fanout tree [37].

The paper is organized as follows. The next subsection provides a brief introduction to alphabetic trees and the concept of enumeration and optimization. In section 2 and 3, some relevant results on alphabetic tree enumeration and algorithms for alphabetic tree optimization, respectively, are presented. In section 4, we introduce the alphabetic fanout optimization problem and a set of rules which increase the effectiveness of our algorithm. We also describe the implementation and present our experimental results in this section. Concluding remarks and future directions are presented in section 5.

1.3 Introduction to Alphabetic Trees

Tree optimization seeks to generate the best tree for a given application. If there are n nodes in a tree where each node i has a weight w_i, a measure of the quality of this tree can be obtained by defining a *tree cost function* in terms of the weights w_i and parameters associated with the tree structure. Using this tree cost function, one tree structure can be compared with another.

Pioneering work in tree optimization was done by Huffman [18]. He assumed that only leaf nodes were weighted. Weight of an internal node is obtained from weights of its immediate children using the weight *combining function*. For example, if nodes i and j with weights w_i and w_j are combined as children of node k, then $w_k = F(w_i, w_j)$, where F denotes the combining function. The application he was addressing was that of generating a prefix-free binary encoding of a set of symbols with minimum average codeword length given the probability of occurrence of each symbol, i.e., generating an optimal binary tree minimizing $\sum_{i=1}^{n} w_i l_i$ where n is the number of leaf nodes, w_i is the weight of leaf node i, and l_i is the length of the path from leaf node i to the root of the tree. The corresponding combining function was $F(w_i, w_j) = w_i + w_j$. Huffman also proposed generalization of the algorithm to generate optimal t-ary trees. It was later discovered that Huffman's algorithm generates optimal trees not only for the *additive* combining function given above, but also for other combining functions (e.g., the *minimax* combining function where $F(w_i, w_j) = max(w_i, w_j) + 1$ and the tree cost function is $max_i(w_i + l_i)$). Glassey and Karp [9] provided the necessary and sufficient conditions for a combining function to generate optimal trees using Huffman's algorithm. Parker [20] provided a more precise characterization of combining functions as *quasi linear* functions and showed that these functions always generate optimal trees under Huffman's algorithm when the tree cost function is *schur* concave. Use of Huffman's algorithm for several applications has been reported in [30, 5, 6, 27].

Variations to tree optimization, e.g., generation of optimal trees with height constraint, generation of optimal trees minimizing the variance of tree depths, generation of optimal trees given an order on the leaf nodes etc., have been proposed for different applications. Trees generated under an order restriction on the leaf nodes are known as *alphabetic trees*. This name was coined in a paper by Gilbert and Moore [8] where they introduced the concept of alphabetic trees in the context of encoding an alphabet with a linear ordering relationship between the letters of the alphabet. Subsequently, alphabetic trees have found applications in fields of computer science (search trees, information storage and retrieval etc.) [24, 17, 7,

40, 19, 39, 12, 11, 3], matrix multiplication [16], information theory [8, 14, 29], fault diagnosis (and similar applications like medical diagnosis, laboratory analysis, genetic identification etc.) [6, 30, 42], and logic synthesis [32, 38].

Pioneering work in tree enumeration of alphabetic trees was done by Cayley in 1889 [2] where he provided the number of alphabetic trees on n leaf nodes. He calculated the number of alphabetic trees while allowing internal nodes to have one or more children. He also considered a case where each internal node bifurcates, i.e., where the resulting alphabetic tree is binary, and rediscovered the *catalan* numbers which give the number of alphabetic binary trees on $n+1$ leaf nodes. Catalan numbers were again derived by Gilbert and Moore in their classic paper [8] where they provided an $O(n^3)$ algorithm to obtain the best binary alphabetic tree. After these initial works, the tree enumeration for alphabetic trees has been left unaddressed. Specifically, the problem of enumerating all alphabetic trees on n leaf nodes where each internal node at least bifurcates, has been left unresolved. This is an important problem as all applications of alphabetic trees listed above require that each internal node at least bifurcates.

The field of tree optimization of alphabetic trees has been more active. Knuth [24] improved upon $O(n^3)$ algorithm of Gilbert and Moore for constructing an optimal alphabetic binary tree by proposing an $O(n^2)$ algorithm utilizing the concept of *monotonicity*. Hu and Tucker [17] proposed an algorithm similar to Huffman's algorithm with run time of $O(nlogn)$. Hu et al. [15] defined a class of combining functions called *regular functions* and showed that all combining functions which are regular functions generate optimal alphabetic binary trees using their algorithm. Kirkpatrick and Klawe [21] proposed a linear algorithm for alphabetic binary tree optimization under minimax combining function with integer leaf weights and an $O(n\log n)$ algorithm for real leaf weights. Wessner [40] and Itai [19] proposed $O(n^2h)$ algorithms to generate optimal alphabetic binary trees with a height restriction h.

For multi-way (i.e., non-binary) alphabetic trees, the work has mainly focused on additive or minimax cost functions. Gotlieb [11] and Vaishnavi et al. [39] independently provided an $O(n^3\log t)$ algorithm for generating optimal alphabetic trees where each internal node has at most t children. Vaishnavi et al. also identified a tree cost function for which $O(n^3t)$ is the best possible runtime and showed that $O(n^3\log t)$ complexity is possible only for a restricted class of tree cost functions. These approaches were proposed for the additive combining function (except for Coppersmith et al. [3], who have addressed the minimax combining function proposing an $O(n\log n)$ algorithm for a unit delay based minimax combining function). However, runtimes of

$O(n^3t)$ holds for a larger class of combining functions.

This paper solves the important problem of counting the number of alphabetic trees where each internal node at least bifurcates. Recurrence equations for the number of alphabetic trees with bounded height or bounded degree or both are also derived. Next, tree optimization algorithms for alphabetic trees with general tree cost functions are provided and it is shown that, with appropriate restrictions, these algorithms reduce to the best known algorithms proposed for additive or minimax tree cost functions. Finally, the application of the tree optimization algorithms to the fanout optimization problem in logic synthesis is presented.

2 Alphabetic Tree Enumeration

2.1 Terminology and Notation

We follow the standard definitions of graphs and trees [4][3]. Trees are connected graphs with no cycles. Weighted graphs refer to node-weighted graphs. A forest refers to a set of trees built on some set of leaf nodes such that each tree is node-disjoint from any other tree and the set of trees in the forest cover the set of leaf nodes. Support of a forest F refers to the set of leaf nodes corresponding to the forest F. The Notation used in this paper is as follows. A k-tree forest with support $\{L_i, \ldots, L_{i+d}\}$ is a set of k trees such that support of these k trees partition $\{L_i, \ldots, L_{i+d}\}$. This k-tree forest is denoted by $F_{k:i,i+d}$. A 1-tree forest on leaf node i through $i+d$, namely, $F_{1:i,i+d}$ is denoted as $T_{i,i+d}$. A 1-tree forest on leaf nodes i through $i+d$ with a height bound h, a degree bound t and a root degree bound r (where $r \leq t$) is denoted by $T^{h:r,t}_{i,i+d}$.

A collection of alphabetic forests is denoted by Ψ. Specifically, a collection of k-tree forests on leaf nodes i through $i+d$ is denoted by $\Psi_{k:i,i+d}$. A collection of 1-tree forests on leaf nodes i through $i+d$, namely $\Psi_{1:i,i+d}$, is also denoted by $\Psi_{i,i+d}$. A collection of 1-tree forests on leaf nodes i through $i+d$ with a height bound h, a degree bound t and a root degree bound r (where $r \leq t$) is denoted by $\Psi^{h:r,t}_{i,i+d}$. The corresponding number of 1-tree forests in these collections of forests on d leaf nodes is denoted by Φ_d. For example, $|\Psi^{h:r,t}_{i,i+d}| = \Phi^{h:r,t}_d$. A collection of 1-tree forests on leaf nodes i through $i+d$ with an *exact* height restriction h, a degree bound t, and a root degree bound r (where $r \leq t$) is denoted by $\Psi^{[h]:r,t}_{i,i+d}$. With this notation, $\Psi^{h:t,t}_{i,i+d}$ corresponds

[3]Note that we had to resort to rather complicated notations to capture the theory in this section. A summary of theoretical results in Sections 2 and 3 is provided in Section 3.5

Figure 1: Illustration of operations JOIN and SPLIT

to collection of alphabetic 1-tree forests with height restriction h and degree restriction t. We denote this as $\Psi_{i,i+d}^{h:t}$. Noting that $\Psi_{i,i+d}^{i:j,k} = \Psi_{i,i+d}^{d:d+1,d+1} \quad \forall i \geq d$ and $\forall j,k \geq d+1$, we replace each index for which there is no restriction by ∞, i.e., on leaf nodes i through $i+d$, collection of 1-tree forests without height restriction is denoted by $\Psi_{i,i+d}^{\infty:t}$ while collection of 1-tree forests without degree restriction is denoted by $\Psi_{i,i+d}^{h:\infty}$. A collection of 1-tree forests on leaf nodes i through $i+d$, namely, $\Psi_{i,i+d}$ is the same as $\Psi_{i,i+d}^{\infty:\infty}$. Finally, $\Psi_{i,i}$ gives the set of alphabetic 1-tree forests on a single leaf node, i.e., the leaf node itself.

2.2 Enumeration

Definition 2.1 *Given a forest F with rooted trees,* JOIN *of F (denoted by $\wedge F$) is obtained by generating a rooted 1-tree forest T with the roots of trees in F as children of the root of the tree in 1-tree forest T (Figure 1).*

Definition 2.2 *Given a rooted 1-tree forest T,* SPLIT *of T (denoted by $\dot{\div}.T$) is obtained by deleting the root making all its children rooted trees in a forest F (Figure 1).*

For example, consider a forest with 4 trees, i.e., $F_4 = \{T_1, T_2, T_3, T_4\}$ as shown in Figure 1. Applying JOIN operation to F_4 results in a 1-tree forest $F_1 = \{T\}$. By applying SPLIT on F_1 we get back $F_4 = \{T_1, T_2, T_3, T_4\}$. Thus, $\wedge(\dot{\div}.F_1) = F_1$ and $\dot{\div}.(\wedge F_4) = F_4$.

These operations are also applicable to a collection of forests. Given a collection of forests $\Psi_{i,i+d}$, $\wedge\Psi_{i,i+d}$ results in a collection of 1-tree forests such that each 1-tree forest corresponds to JOIN of some forest in $\Psi_{i,i+d}$. Likewise, given a collection of 1-tree forests $\Psi_{i,i+d}$, $\dot{\div}.\Psi_{i,i+d}$ results in a collection of forests such that each forest in this collection corresponds to SPLIT of some 1-tree forest in $\Psi_{i,i+d}$. For example, if $\Psi = \{\{T_{i,i+j'}, T_{i+j'+1,i+k'}, T_{i+k'+1,i+d}\}, \{T_{i,i+j''}, T_{i+j''+1,i+k''}, T_{i+k''+1,i+d}\}\}$, where $j' < k' < d$ and $j'' < k'' < d$, then $\wedge\Psi = \Psi' = \{\{T'_{i,i+d}\}, \{T''_{i,i+d}\}\}$ where $\{T'_{i,i+d}\} =$

Figure 2: A graphical illustration of one-to-one correspondence between $\Psi_{i,i+3}$ and $\bigcup_{k=2}^{4} \Psi_{k:i,i+3}$

$\wedge\{T_{i,i+j'}, T_{i+j'+1,i+k'}, T_{i+k'+1,i+d}\}$ and $\{T''_{i,i+d}\} = \wedge\{T_{i,i+j''}, T_{i+j''+1,i+k''}, T_{i+k''+1,i+d}\}$ and $.\doteq.\Psi' = \Psi$. SPLIT can also be applied on a collection of forests where each forest has different number of trees.

For the purpose of alphabetic tree enumeration, all distinct alphabetic trees on leaf nodes 1 through n need to be counted. In the rest of the paper, we use $\Psi_{1,n}$ to refer to this exhaustive collection of trees on n leaf nodes and Φ_n to refer to the number of such trees, i.e., $\Phi_n = |\Psi_{1,n}|$.

Lemma 2.1

$$\Psi_{i,i+d} = \bigcup_{k=2}^{d+1} \wedge(\Psi_{k:i,i+d}) \tag{1}$$

Proof $\Psi_{i,i+d}$ could be partitioned with respect to the arity of the root of each element of $\Psi_{i,i+d}$. The lemma follows from the observation that there is a one to one correspondence between each element of $\Psi_{k:i,i+d}$ and elements of $\Psi_{i,i+d}$ with k-ary roots. This is also shown in Figure 2. ■

Corollary 2.2

$$.\doteq.\Psi_{i,i+d} = \bigcup_{k=2}^{d+1} \Psi_{k:i,i+d} \tag{2}$$

Definition 2.3 *Given two sets Ψ' and Ψ'' we define their* Cartesian product *(denoted by $\times$) as:* $\Psi' \times \Psi'' = \{F \mid F = F' \cup F'',\ F' \in \Psi',\ F'' \in \Psi''\}$.

Theorem 2.3

$$\begin{aligned}\Psi_{i,i+d} &= \bigcup_{j=0}^{d-2} (\wedge(\Psi_{i,i+j} \times \Psi_{i+j+1,i+d}) \cup \wedge(\Psi_{i,i+j} \times (.\doteq.\Psi_{i+j+1,i+d}))) \\ &\quad \bigcup \wedge(\Psi_{i,i+d-1} \times \Psi_{i+d,i+d}) \qquad\qquad for\ d \geq 1 \end{aligned} \tag{3}$$

Proof First, we show that

$$\Psi_{i,i+d} = \bigcup_{j=0}^{d-1} \wedge(\Psi_{i,i+j} \times [\bigcup_{k=1}^{d-j} \Psi_{k:i+j+1,i+d}]) \qquad \text{for } d \geq 1 \tag{4}$$

Examining equation (4), for some values of j and k, we are JOINing a 1-tree forest with support $\{i, \ldots, i+j\}$ with a k-tree forest with support $\{i+j+1, \ldots, i+d\}$ to generate a 1-tree forest with support $\{i, \ldots, i+d\}$. Increasing value of j in this equation corresponds to incrementally adding more leaf nodes to the leftmost subtree of the tree root. Denote by $\Psi_{(i,i+j),i+d}$ the collection of alphabetic trees on leaf nodes i through $i+d$ such that $0 \leq j < d$ and leaf node $i+j$ is the rightmost leaf node of the leftmost subtree. Hence,

$$\Psi_{i,i+d} = \bigcup_{j=0}^{d-1} \Psi_{(i,i+j),i+d} \tag{5}$$

A generalization of lemma 2.1 gives us:

$$\Psi_{(i,i+j),i+d} = \bigcup_{k=2}^{d-j+1} \wedge(\Psi_{k:(i,i+j),i+d}) \tag{6}$$

However, all k-ary trees with node $i+j$ as the rightmost leaf node of the leftmost tree can be generated by JOINing each tree on leaf nodes i through $i+j$ with all possible $k-1$ rooted forests on leaf nodes $i+j+1$ through $i+d$. This allows us to rewrite equation (6) as:

$$\begin{aligned}\Psi_{(i,i+j),i+d} &= \bigcup_{k=2}^{d-j+1} \wedge(\Psi_{i,i+j} \times \Psi_{k-1:i+j+1,i+d}) \\ &= \bigcup_{k=1}^{d-j} \wedge(\Psi_{i,i+j} \times \Psi_{k:i+j+1,i+d}) \end{aligned}$$

Substituting the above in equation (5) we get:

$$\begin{aligned}\Psi_{i,i+d} &= \bigcup_{j=0}^{d-1} \bigcup_{k=1}^{d-j} \wedge(\Psi_{i,i+j} \times \Psi_{k:i+j+1,i+d}) \\ &= \bigcup_{j=0}^{d-1} \wedge(\Psi_{i,i+j} \times [\bigcup_{k=1}^{d-j} \Psi_{k:i+j+1,i+d}]) \end{aligned}$$

Figure 3: Generating alphabetic trees for a) 1 leaf node, b) 2 leaf nodes, c) 3 leaf nodes, and d) 4 leaf nodes

The second term of the Cartesian product above corresponds to all k-tree forests for $1 \leq k \leq d-j$ on $d-j$ leaf nodes (leaf nodes $i+j+1$ through $i+d$). All possible k-tree forests on leaf nodes $i+j+1$ through $i+d$ for $2 \leq k \leq d-j$, namely $\Psi_{k:i+j+1,i+d}$, can be obtained by SPLITting each element of $\Psi_{i+j+1,i+d}$ with k-ary root. This gives the following $d-1$ equations.

$$\begin{aligned}
\Psi_{2:i,i+d} &= \Psi_{i,i} \times \Psi_{i+1,i+d} \cup \cdots \cup \Psi_{i,i+d-2} \times \Psi_{i+d-1,i+d} \cup \Psi_{i,i+d-1} \times \Psi_{i+d,i+d} \\
\Psi_{3:i,i+d} &= \Psi_{i,i} \times \Psi_{2:i+1,i+d} \cup \cdots \cup \Psi_{i,i+d-2} \times \Psi_{2:i+d-1,i+d} \\
&\vdots \qquad\quad \vdots \\
\Psi_{d+1:i,i+d} &= \Psi_{i,i} \times \Psi_{d:i+1,i+d}
\end{aligned}$$

Taking the union of RHS's and then JOINing each element gives us $\Psi_{i,i+d}$. Performing the same operation on LHS's and applying corollary 2.2 on terms of the last $d-2$ equations above gives us the desired result. ■

Equation (3) is illustrated for n = 1, 2, 3 and 4 in Figure 3. From (3), Φ_n for n leaf nodes is determined using a recursive difference equation given below. Detailed derivation is omitted to save space.

$$\Phi_n = \begin{cases} 2\sum_{j=1}^{n-2} (\Phi_{n-j}\Phi_j) + \Phi_{n-1}\Phi_1 \text{for } n \geq 2 \\ 1 \qquad\qquad \text{for } n = 1 \end{cases} \tag{7}$$

The above equation is a convoluted recurrence equation whose solution is given below.

Lemma 2.4 *The number of alphabetic trees on n leaf nodes is given by:*

$$\Phi_n = -\frac{1}{4}\sum_{j=0}^{\lfloor n/2 \rfloor} \binom{n-j}{j} (-6)^{n-2j} \binom{1/2}{n-j} \text{for } n \geq 2 \tag{8}$$

Detailed proof is omitted to save space.

This sequence is identical to the sequence generated for dissection of a polygon by Motzkin [28]. The same sequence is also derived in [34] while solving equations for a pair of inverse series without reference to any particular application. The formula proposed there is given below:

$$\Phi_n = \sum_{k=0}^{n} (-1)^j \binom{2n-j}{j} \binom{2n-2j}{n-j} \frac{2^{n-j}}{n-j+1} \tag{9}$$

Indeed, we have formally shown that equations (8) and (9) are equivalent. Motzkin has also shown that for large n $\Phi_{n+1}/\Phi_n \rightarrow 5.83$, implying that Φ_n is $O(6^n)$.

It is interesting to note that if we restrict alphabetic trees to be binary, equation (3) is reduced to:

$$\Psi_{i,i+d} \quad = \quad \bigcup_{j=0}^{d-1} \wedge(\Psi_{i,i+j} \times \Psi_{i+j+1,i+d}) \qquad\qquad \text{for } d \geq 1$$

This gives rise to a simpler convoluted recurrence equation of the form $\Phi_n = \sum_{j=1}^{n-1} \Phi_{n-j}\Phi_j$ for $n \geq 1$ whose solution gives the Catalan numbers. Thus, the number of alphabetic binary trees on n leaf nodes is given by:

$$\binom{2n-2}{n-1} \frac{1}{n} \approx \frac{4^{n-1}}{\sqrt{\pi}(n-1)^{3/2}} \qquad \text{for large } n. \tag{10}$$

Cayley [2], and Gilbert and Moore [8] have independently derived expressions for the number of alphabetic binary trees which are equivalent to the Catalan numbers.

2.3 Bounded Height Trees

Height of a tree is the maximum number of edges on the path from root to any leaf node. Let us now enumerate alphabetic trees with height restriction h. The tree enumeration equation for bounded height alphabetic trees is derived from equation (3) as:

$$\begin{aligned}\Psi_{i,i+d}^{h:\infty} &= \bigcup_{j=0}^{d-2} (\wedge(\Psi_{i,i+j}^{h-1:\infty} \times \Psi_{i+j+1,i+d}^{h-1:\infty}) \cup \wedge(\Psi_{i,i+j}^{h-1:\infty} \times (.\dot{=}.\Psi_{i+j+1,i+d}^{h:\infty}))) \\ &\quad \bigcup \wedge(\Psi_{i,i+d-1}^{h-1:\infty} \times \Psi_{i+d,i+d}^{h-1:\infty}) \qquad \text{for } d \geq 1 \end{aligned} \tag{11}$$

Correctness of the equation follows from the observation that alphabetic trees with maximum height h can be generated by JOINing all alphabetic trees with maximum height $h-1$. The number of such trees is computed using the following recursive difference equation:

$$\Phi_n^{h:\infty} = \begin{cases} \sum_{j=1}^{n-2} (\Phi_j^{h-1:\infty}(\Phi_{n-j}^{h-1:\infty} + \Phi_{n-j}^{h:\infty})) + \Phi_{n-1}^{h-1:\infty} & \text{for } n \geq 2; h < n \\ 1 \qquad \text{for } n = 1 \text{ or } h = 1 \end{cases} \tag{12}$$

Unlike equation (7), equation (12) is a multi-variable recurrence equation for which no closed form solutions exists. (This also applies to equation (15) and (17) in Section 2.4 and 2.5, respectively.) However, equation (12) can easily be used to derive the number of trees given the number of leaf nodes and a height constraint.

Using $\Phi_n^{h:\infty}$ and $\Phi_n^{h-1:\infty}$, the number of alphabetic trees with exact height h can also be derived, i.e., $\Phi_n^{[h]:\infty} = \Phi_n^{h:\infty} - \Phi_n^{h-1:\infty}$ with the initial condition $\Phi_n^{[1]:\infty} = \Phi_n^{1:\infty}$.

Maximum height of a tree with n leaf nodes is $n-1$. Hence, the number of alphabetic trees on n leaf nodes is also given by:

$$\sum_{h=1}^{n-1} \Phi_n^{[h]:\infty} = \sum_{h=1}^{n-1} (\Phi_n^{h:\infty} - \Phi_n^{h-1:\infty}) + \Phi_n^{1:\infty} = \Phi_n^{n-1:\infty} \tag{13}$$

2.4 Bounded Degree Trees

Some applications require that internal nodes have no more than t children. The corresponding enumeration equation is:

$$\Psi_{i,i+d}^{\infty:r,t} = \bigcup_{j=0}^{d-2} (\wedge(\Psi_{i,i+j}^{\infty:t} \times \Psi_{i+j+1,i+d}^{\infty:t}) \cup \wedge(\Psi_{i,i+j}^{\infty:t} \times (.\dot{=}.\Psi_{i+j+1,i+d}^{\infty:r-1,t})))$$

$$\bigcup \wedge(\Psi_{i,i+d-1}^{\infty:t} \times \Psi_{i+d,i+d}^{\infty:t}) \qquad \text{for } d \geq 1; \quad 2 \leq r \leq t \leq d+1 \tag{14}$$

Note that $\Psi_{i,i+d}^{\infty:r,t}$ = NULL set for $r = 1$ or $t = 1$.

Hence:

$$\Phi_n^{\infty:r,t} = \begin{cases} \sum_{j=1}^{n-2} (\Phi_j^{\infty:t}(\Phi_{n-j}^{\infty:t} + \Phi_{n-j}^{\infty:r-1,t})) + \Phi_{n-1}^{\infty:t} & \text{for } n \geq 2; \;\; 2 \leq r \leq t \\ 0 & \text{for } r = 1 \\ 1 & \text{for } n = 1 \text{ and } r \neq 1 \end{cases} \tag{15}$$

2.5 Bounded Height and Bounded Degree Trees

In the most general setting, alphabetic trees with both degree and height restriction can be enumerated. Corresponding enumeration equation and recurrence equations are as given below.

$$\begin{aligned} \Psi_{i,i+d}^{h:r,t} \quad = \quad & \bigcup_{j=0}^{d-2} (\wedge(\Psi_{i,i+j}^{h-1:t} \times \Psi_{i+j+1,i+d}^{h-1:t}) \cup \wedge(\Psi_{i,i+j}^{h-1:t} \times (\dot{-}.\Psi_{i+j+1,i+d}^{h-1:r-1,t}))) \\ & \bigcup \wedge(\Psi_{i,i+d-1}^{h-1:t} \times \Psi_{i+d,i+d}^{h-1:t}) \quad \text{for } d \geq 1; 2 \leq r \leq t \leq d+1; h < n \end{aligned} \tag{16}$$

Again, $\Psi_{i,i+d}^{h:r,t}$ = NULL set for $r = 1$ or $t = 1$.

$$\Phi_n^{h:r,t} = \begin{cases} \sum_{j=1}^{n-2} (\Phi_j^{h-1:t}(\Phi_{n-j}^{h-1:t} + \Phi_{n-j}^{h:r-1,t})) + \Phi_{n-1}^{h-1:t} \text{ for } n \geq 2; 2 \leq r \leq t; h \geq 2 \\ 0 \qquad \text{for } r = 1 \\ 1 \qquad \text{for } h = 1 \text{ or } n = 1 \text{ when } r \neq 1 \end{cases} \tag{17}$$

It should be noted that solution to above equation will automatically provide solutions to equation (7), equation (12) and equation (15) as special cases.

3 Alphabetic Tree Optimization

Alphabetic tree optimization implies finding the "best" alphabetic tree given a combining function and a tree cost function. Formally, the problem of alphabetic tree optimization can be defined as given below. Here, without loss of generality, we assume that the tree cost function is to be minimized.

Problem 3.1 ALPHABETIC_TREE_OPTIMIZATION

Instance: *A set of n ordered and weighted leaf nodes $(L_1, L_2, \ldots, L_n)$ with corresponding weights W_{L_i} and a combining function F which combines $t \geq 2$ nodes to generate an internal node I_p with weight $W_{I_p} = F(W_{child_{I_{p,1}}}, \ldots, W_{child_{I_{p,t}}})$ and a tree cost function $C_T = C(W_{L_1}, \ldots, W_{L_n})$.*

Problem: *Generate a minimum cost tree that has no internal edge crossing.*

The following algorithm enumerates all alphabetic trees and selects the best alphabetic tree with respect to the given tree cost function.

Algorithm 2.1 GenBestAlpTree ($\mathcal{N}$)
$\mathcal{N}$ is a set of n leaf nodes with weights
begin
1 **for** $d = 0$ to $n-1$ **do**
2 **for** $i = 1$ to $n-d$ **do**
3 **if** $d = 0$ $\Psi_{i,i}$ =SingleNodeTree(i)
4 **else**
5 **for** $l = 1$ to d $\Psi_{l:i,i+d}$ = NULL
6 **for** $j = 0$ to $d-1$ **do**
7 **ForEachElement** $F_{1:i,i+j}$ of $\Psi_{1:i,i+j}$ **do**
8 **for** $l = 1$ to $d-j$ **do**
9 **ForEachElement** $F_{l:i+j+1,i+d}$ of $\Psi_{l:i+j+1,i+d}$ **do**
10 $\Psi_{l+1:i,i+d} = \Psi_{l+1:i,i+d} \bigcup \{F_{1:i,i+j} \cup F_{l:i+j+1,i+d}\}$
11 $\Psi_{i,i+d} = \Psi_{i,i+d} \bigcup \{\wedge(F_{1:i,i+j} \cup F_{l:i+j+1,i+d})\}$
12 Result = FindBest($\Psi_{1,n}$)
end

Here d corresponds to the number of leaf nodes being considered in the main loop of the algorithm, and i corresponds to the first leaf node of the set of leaf nodes being considered in the current loop. Thus, $i + d$ corresponds to the last leaf node in the set of leaf nodes being considered in the current loop. j corresponds to the number of leaf nodes under the leftmost branch. Thus, $d-j$ corresponds to the number of leaf nodes under all other branches of the tree.

l corresponds to the number of siblings of the leftmost branch. The braces used in line 10 signify a set generation operation. Line 10 results in $l+1$-tree forests by generating a set from the union of 1-tree forests on leaf nodes i to $i+j$ with l-tree forests on leaf nodes $i+j+1$ to $i+d$. Line 11 generates the corresponding 1-tree forest by JOINing these $l+1$-tree forest. The algorithm can be easily modified to generate optimal alphabetic trees with each internal node having at most t children by restricting $l < t$. With small modifications, a generic algorithm for bounded-height and bounded-degree alphabetic tree enumeration can be obtained using equation (17).

Lemma 3.1 *The time and space complexity of algorithm 2.1 are* $O(n^3 6^{n+1})$ *and* $O(n^3 6^n)$, *respectively.*[4]

Proof Motzkin [28] has shown that for large n, $\Phi_{n+1}/\Phi_n \to 5.83$. Hence, Φ_n is $O(6^n)$. In line 7 of the algorithm, there are $O(6^{j+1})$ elements in $\Psi_{1:i,i+j}$. Likewise, there are totally $O(6^{d-j})$ elements in the right branches of the tree (corresponding to line 8, 9, 10 and 11). Thus, the total time complexity of inner loop corresponding to line 7-11 is $O(6^{d+1})$. Hence, the time complexity can be calculated as:

$$\sum_{d=1}^{n-1}[\sum_{i=1}^{n-d}(\sum_{j=0}^{d-1}6^{d+1}+d)]+n=\sum_{d=1}^{n-1}(n-d)[d6^{d+1}+d^2]=O(n^3 6^{n+1})$$

Since there are $O(n^2)$ ordered subsets of leaf nodes (e.g., $n-1$ two leaf node sets, $n-2$ three leaf node sets, etc..), assuming that the information about an n leaf tree can be stored in $O(n)$ space, a naive implementation of this algorithm requires space complexity $O(n^3 6^n)$ for maintaining all alphabetic forests for every ordered subset of leaf nodes. ■

In general, depending on the combining and tree cost functions, complexity of determining the best tree may be reduced by considering only a subset of all tree structures which are non-inferior with respect to each other for the purpose of optimizing the tree cost. In particular, if the tree is *subtree optimal*, as will be defined next, the optimal alphabetic binary trees can be found in polynomial time for arbitrary tree cost functions. Similarly, the optimal alphabetic non-binary tree can be found in polynomial time if the tree is *subforest optimal* as defined next.

[4]These bounds are not tight.

3.1 Subtree and Subforest Optimality

Given a set of internal nodes $\mathcal{I} = \{I_1, I_2, \ldots, I_m\}$ and leaf nodes $\mathcal{L} = \{L_1, L_2, \ldots, L_n\}$ of a tree where I_m is the root of the tree, cost of the tree is given by $C_T = C(W_{L_1}, \ldots, W_{L_n})$ where W_N denotes weight of leaf node N. The objective is to solve problem 3.1. Consider two trees T and T' on leaf nodes $\{L_1, \ldots, L_n\}$. Let us assume that these trees only differ in subtrees rooted at internal node I, that is, T and T' are identical except for differences between subtrees T_I and T'_I. Then, the tree is **subtree optimal** (ST-optimal) if we can define a *subtree cost function* H for every such T_I and T'_I such that; $H_{T_I} > H_{T_{I'}} \Rightarrow C_T \geq C_{T'}$.

Definition 3.1 *A tree is ST-optimal if the tree* cost *is monotone non-decreasing in subtree cost of each of its subtrees, that is, if we increase (decrease) the subtree cost of some subtree, the tree cost will not decrease (increase). Necessary conditions for ST-optimality are:*

1. *The tree cost function C_T is decomposable in terms of subtree cost function of each of its subtrees, i.e., $C_T = G(H_{T_I}, W_{\mathcal{L}}) \quad \forall I \in \mathcal{I}$.*

2. *Function G is independent of the tree structure T_I at node I.*

3. *Function G is monotone non-decreasing in H_{T_I}.*

This allows us to independently optimize $H_{T_I} \quad \forall I \in \mathcal{I}$ using dynamic programming. Hence, to determine whether the given tree optimization problem is ST-optimal, a function H_T satisfying the above conditions needs to be identified. Fortunately, for most ST-optimal trees, the tree cost is easily decomposable in terms of the tree cost of its subtrees (i.e., $H_T = C_T$). In Figure 4, let $\{L_1, L_2, L_3, L_4, L_5\}$ be the set of leaf nodes with weights $\{W_{L_1}, W_{L_2}, W_{L_3}, W_{L_4}, W_{L_5}\}$. Two alphabetic binary tree structures are shown in the figure. Both additive tree cost and minimax tree costs allow us to decompose the tree cost in terms of tree costs of its subtrees, resulting in ST-optimality of the additive and minimax tree cost functions.

The implication of ST-optimality is that if a tree is ST-optimal, with a dynamic programming based approach, only the optimal subtrees for each subset of leaf nodes need to be maintained. This is sufficient to reduce the time complexity of alphabetic binary tree optimization from exponential to polynomial as described in section 3.2. However, for non-binary trees, this is not sufficient to reduce exponential time complexity to polynomial. Polynomial runtime is achieved if the tree is subforest optimal as described next.

Figure 4: Examples of ST-optimal tree cost functions

Let us consider an internal node I of a tree T with V immediate children and leaf support $\{L_i, \ldots, L_{i+d}\}$. Consider any subset $\mathcal{D}$ of these V children such that these children have continuous leaf support $\{L_{i+j}, \ldots, L_{i+k}\}$ where $0 \leq j \leq k \leq d$. Let $D = |\mathcal{D}|$ where $1 \leq D \leq V$. Let us denote the corresponding tree structures as $T_1, T_2, \ldots, T_D$ with tree costs $C_{T_1}, C_{T_2}, \ldots, C_{T_D}$, respectively. We define this set of trees as a **subforest** of the original tree T. Now, let us consider another tree T' that has identical tree structure except for the tree structures rooted at these D children of internal node I. Let us denote the corresponding tree structures as $T'_1, T'_2, \ldots, T'_D$ with tree costs $C_{T'_1}, C_{T'_2}, \ldots, C_{T'_D}$. Then, the tree is **subforest optimal** (SF-optimal) if we can define a *subforest cost function* H such that; $H(C_{T_1}, C_{T_2}, \ldots, C_{T_D}) > H(C_{T'_1}, C_{T'_2}, \ldots, C_{T'_D}) \Rightarrow C_T \geq C_{T'}$.

Definition 3.2 *A tree is SF-optimal if the tree cost is monotone non-decreasing in subforest cost of each of its subforests, that is, if we increase (decrease) the subforest cost of some subforest, the tree cost will not decrease (increase). Necessary conditions for SF-optimality are:*

1. *The tree cost function C_T is decomposable in terms of the subforest cost function of each of its subforests, i.e., $C_T = G(H(C_{T_1}, C_{T_2}, \ldots, C_{T_D}), W_{\mathcal{L}})$ for all subforests $\mathcal{D}$ of T.*

2. *Function G is independent of the subforest structure of $\mathcal{D}$.*

3. Function G is monotone non-decreasing in $H(C_{T_1}, C_{T_2}, \ldots, C_{T_D})$.

SF-optimality is a generalization of ST-optimality with ST-optimality being a special case of SF-optimality when we have $D = 1$. Indeed, a notion of strong SF-optimality can be proposed that distinguishes between two forests of any number of trees on the same set of leaf nodes. This is however, outside the scope of this paper.

It is interesting to note here that conditions of SF-optimality are subsumed by the *principle of optimality* (see [26], for example) used in characterizing the decomposable problems for the dynamic programming approach. Hence, conditions for SF-optimality can be viewed as specialization of the principles of optimality for tree optimization.

Examples of SF-optimal trees include the original Huffman trees with the additive cost function as shown in Figure 5. Minimax trees are also SF-optimal. A variation of Minimax trees, namely, trees with combining function $F(W_1, \ldots, W_n) = Max(W_1, \ldots, W_n) + n$ are SF-optimal. These minimax trees are very important in many applications, specially in logic synthesis where this latter combining function corresponds to the unit fanout delay model. SF-optimality of unit fanout delay model has allowed us to propose an optimal alphabetic fanout tree in polynomial time. An example of the minimax combining function with unit fanout delay model is shown in Figure 5.

Note that ST-optimality (or SF-optimality) is a property of the tree optimization problem being solved. Indeed, the combining functions and the tree cost functions determine ST-optimality or SF-optimality of trees generated and the underlying tree optimization problem. It should also be pointed out that ST-optimality and SF-optimality apply to alphabetic as well as non-alphabetic trees. In essence, ST-optimality characterizes binary trees that permit the use of dynamic programming approach while guaranteeing time complexity of $O(n^3)$ or better while strong SF-optimality characterizes non-binary trees that permit use of dynamic programming approach while guaranteeing time complexity of $O(n^3)$ or better. Trees which are SF-optimal but are not strongly SF-optimal allow use of dynamic programming approach while guaranteeing time complexity of $O(n^4)$ or better. We discuss the time and space complexity issues for ST-optimal and SF-optimal trees next.

3.2 ST-optimal Trees

Figure 5: SF-optimal tree cost functions

Figure 6: A graphical illustration of ST-optimal alphabetic tree generation equation

Lemma 3.2 *Consider a tree optimization problem. If the resulting trees are ST-optimal, then there exists an optimal tree with optimal subtrees.*

Proof Since the tree is ST-optimal, we could always substitute any non-optimal subtree by an optimal subtree without increasing the tree cost. ■

Corollary 3.3 *To generate an optimal alphabetic tree using equation (3), it is sufficient to consider only optimal trees as arguments to the* JOIN *operator.*

Let $\theta_{i,i+j}$ denote the optimal tree on the set of leaves $(i, \ldots, i+j)$ and $St\Psi_{i+j+1,i+d}$ denote the collection of alphabetic trees on leaves $(L_{i+j+1}, \ldots, L_{i+d})$ such that for each tree in $St\Psi_{i+j+1,i+d}$, every subtree is optimal. Then, equations (3) and (7) can be rewritten as:

$$\begin{aligned} St\Psi_{i,i+d} \quad = \quad & \bigcup_{j=0}^{d-2} \Big((\wedge\{\theta_{i,i+j}, \theta_{i+j+1,i+d}\}) \cup (\wedge(\{\theta_{i,i+j}\} \times (\dot{\div}.St\Psi_{i+j+1,i+d}))) \Big) \\ & \bigcup (\wedge\{\theta_{i,i+d-1}, \theta_{i+d,i+d}\}) \qquad \text{for } d \geq 1 \end{aligned} \tag{18}$$

$$St\Phi_n = \begin{cases} \sum_{j=1}^{j-2} [St\Phi_{n-d} + 1] + 1 & \text{if } n \geq 2 \\ 1 \qquad \text{if } n = 1 \end{cases} \tag{19}$$

Lemma 3.4 *The number of ST-optimal alphabetic trees on n leaves ($n \geq 2$), $St\Phi_n$, is equal to* $2^{n-1} - 1$.

Proof We prove this by induction.

- For two leaf nodes $St\Phi_2 = 2^{2-1} - 1 = 1$ which is true since there is only one tree structure on two leaves.

- Assuming the above equation is true for $i \geq 2$, we prove its correctness for $i + 1$:

$$St\Phi_{i+1} = \sum_{d=1}^{i-1} [St\Phi_{i+1-d} + 1] = \sum_{d=1}^{i-1} [2^{i-d} - 1 + 1] + 1 = \sum_{d=0}^{i-1} [2^d] + 1 - 2^0 = 2^i - 1$$

Thus, $St\Phi_n = 2^{n-1} - 1$. ■

For all practical purposes, $St\Psi_{1,n}$ is the largest collection of trees any optimal alphabetic tree construction algorithm has to consider when the tree is ST-optimal. This leads to an upper bound on the complexity of any ST-optimal alphabetic tree problem. A corresponding algorithm is shown below.

Algorithm 2.2 GenBestAlpTree-STOptimal ($\mathcal{N}$)
$\mathcal{N}$ is a set of n leaf nodes with weights
begin
1 **for** d = 0 to $n - 1$ **do**
2 **for** i = 1 to $n - d$ **do**
3 **if** $d = 0$ $St\Psi_{i,i}$ =SingleNodeTree(i)
4 **else**
5 $\theta_{i,i+d}$ = NULL
6 **for** l = 1 to d $St\Psi_{l:i,i+d}$ = NULL
7 **for** j = 0 to $d - 1$ **do**
8 **for** l = 1 to $d - j$ **do**
9 **ForEachElement** $F_{l:i+j+1,i+d}$ of $St\Psi_{l:i+j+1,i+d}$ **do**
10 $St\Psi_{l+1:i,i+d} = St\Psi_{l+1:i,i+d} \bigcup \{\{\theta_{i,i+j}\} \cup F_{l:i+j+1,i+d}\}$
11 $\theta_{i,i+d}$ = ChooseBest($\theta_{i,i+d}, \wedge(\{\theta_{i,i+j}\} \cup F_{l:i+j+1,i+d})$)
12 Result = $\theta_{1,n}$
end

Algorithm 2.2. is similar to algorithm 2.1. However, algorithm 2.2 does not enumerate through each element of $St\Psi_{i,i+j}$ corresponding to line 7 of algorithm 2.1 because $St\Psi_{1:i,i+j} = \{\{\theta_{i,i+j}\}\}$. Apart from this, in line 11 algorithm 2.2 maintains only the best tree while the corresponding line in algorithm 2.1 maintained a collection of 1-tree forests.

Lemma 3.5 *The time and space complexities of algorithm 2.2 are* $O(2^{n+1})$ *and* $O(n^3 2^{n-1})$, *respectively.*

Proof The exponential time complexity of this algorithm is due to exponentially large ways of selecting optimal subtrees to generate subforests at an internal node, i.e., $|St\Psi_{i+j+1,i+d} \cup \dot{=}.St\Psi_{i+j+1,i+d}| = \sum_{l=1}^{d-j} |St\Psi_{l:i+j+1,i+d}|$. The number of subforests on $d-j-1$ leaf nodes is also given by the number of ways an ordered set of $d-j-1$ elements can be partitioned. This is shown in Figure 7. When the left most branch of the tree is already determined, we have $d-j-1$ ways of generating 3-tree subforests with optimal alphabetic trees, $(d-j-1)(d-j-2)/2$ ways of generating 4-tree subforests with optimal alphabetic trees, etc.. The recurrence equation for this problem is $\Phi'_{d-j} = \sum_{i=1}^{d-j-1} \Phi_i{}' + 1$ with the solution of $\Phi'_{d-j} = 2^{d-j-1}$, giving the runtime of inner most loops as well as the number of subforests to be maintained on every ordered subset of leaf nodes. Hence, the time complexity of algorithm 2.2 is:

$$\sum_{d=1}^{n-1} [\sum_{i=1}^{n-d} (\sum_{j=0}^{d-1} 2^{d-j-1} + d)] + n = 2^{n+1} + 5n^3/6 - 3n^2/2 + 2n/3 - 2 = O(2^{n+1})$$

Again, since there are $O(n^2)$ ordered subsets of leaf nodes, assuming that the information about an n leaf tree can be stored in $O(n)$ space, a naive implementation of this algorithm requires space complexity $O(n^3)$ for maintaining optimal subtrees for every ordered subset of leaf nodes. However, for each ordered subset of leaf nodes, a list of subforests consisting of ST-optimal subtrees needs to be maintained. Since each subtree is optimal, only a list of pointers to corresponding optimal subtrees need to be kept, requiring $O(n)$ space per subforest. As explained above, since there are 2^{n-1} ways to partition an ordered set of n nodes, and since since each partition corresponds to a subforest, the space complexity of the algorithm is $O(n^3 2^{n-1})$. ■

Thus, for non-binary trees, ST-optimality of the trees has improved the run time and space complexity significantly although still not reducing them to polynomial. However, for alphabetic binary trees, ST-optimality is sufficient to reduce the run times and space complexity to polynomial as will be described next.

Figure 7: Example showing 8 possible ways to generate alphabetic trees with four leaf nodes under right branches

3.3 ST-optimal Binary Trees

Lemma 3.6 *The number of ST-optimal alphabetic binary trees on n leaf nodes is equal to* $n-1$.

Proof For ST-optimal binary trees, $\wedge(\{\theta_{i,i+j}\} \times (\dot{=}.St\Psi_{i+j+1,i+d}))$ drops out of equation (18), thus

$$Bi\Psi_{i,i+d} \quad = \quad \bigcup_{j=0}^{d-1} (\wedge\{\theta_{i,i+j}, \theta_{i+j+1,i+d}\}) \qquad \text{for } d \geq 1 \tag{20}$$

where $Bi\Psi_{i,i+d}$ denote the collection of apropos alphabetic binary trees. This gives:

$$\Phi_n = \begin{cases} n-1 & \text{if } n \geq 2 \\ 1 & \text{if } n = 1 \end{cases} \tag{21}$$

■

Lemma 3.6 can be exploited to generate optimal alphabetic binary trees in $O(n^3)$ time complexity as follows.

Algorithm 2.3 GenBestAlpBinaryTree-STOptimal ($\mathcal{N}$)
$\mathcal{N}$ is given set of n leaf nodes with weights
begin
1 **for** d = 0 to $n-1$ **do**
2 **for** i = 1 to $n-d$ **do**
3 $Bi\Psi_{i,i+d}$ = NULL
4 $\theta_{i,i+d}$ = NULL
5 **if** $d = 0$ $Bi\Psi_{i,i}$ =SingleNodeTree(i)
6 **else**
7 **for** k = 0 to $d-1$ **do**
8 $Bi\Psi_{i,i+d} = Bi\Psi_{i,i+d} \bigcup \wedge\{\theta_{i,i+k} \cup \theta_{i+k+1,i+d}\}$
9 $\theta_{i,i+d}$ = ChooseBest$(\theta_{i,i+d}, \wedge\{\theta_{i,i+k} \cup \theta_{i+k+1,i+d}\})$
10 Result = $\theta_{1,n}$
end

For each i, we generate $n-i$ optimal binary trees on subset of i leaf nodes. This has to be done n times; thus, time complexity of algorithm 2.3 is

$$\sum_{i=1}^{i=n-1} i(n-i) = n^3/6 - n/6 = O(n^3).$$

Since for each of the $n(n-1)/2$ ordered subsets, we need to maintain only two pointers (one pointing to the ordered subset which corresponds to the best right subtree and the other pointing to the best left subtree), the space complexity of the algorithm is $O(n^2)$.

As mentioned in section 2, the number of distinct alphabetic binary trees on n leaf nodes is given by the Catalan numbers that are $O(4^n)$. Subtree optimality of tree cost reduces the number of trees to be considered significantly, resulting in a polynomial time algorithm for finding an optimal alphabetic binary tree. This remains true irrespective of other characteristics of weights, tree cost function and combining functions. Previous researchers [17, 15, 8, 23, 21, 3] have generated optimal alphabetic binary trees, restricting leaf weights and/or some parameters of combining function to be integer, or only with respect to a specific tree cost function. Our algorithm generates optimal alphabetic trees for ST-optimal trees in $O(n^3)$ without any such restriction. This run time can be further reduced to $O(n^2)$ for certain tree cost functions

satisfying the *monotonicity* principle proposed by Knuth in [24]. Monotonicity property of a tree cost function guarantees that for optimal alphabetic trees, the tree cost of the left most branch of the root will not decrease if an additional leaf node is introduced to the right of all leaf nodes (and vice versa). The corresponding algorithm is given below.

Algorithm 2.4 GenBestAlpBinaryTree-STOptimalMonotone ($\mathcal{N}$)
$\mathcal{N}$ is given set of n leaf nodes with weights
begin
1 **for** d = 0 to $n-1$ **do**
2 **for** i = 1 to $n-d$ **do**
3 $Bi\Psi_{i,i+d}$ = NULL
4 $\theta_{i,i+d}$ = NULL
5 **if** $d = 0$ $Bi\Psi_{i,i}$ =SingleNodeTree(i)
6 **else**
7 StartIndexOffset = IndexOfRightMostLeafOfLeftSubTree($\theta_{i,i+d-1}$) $-i$
8 StopIndexOffset = IndexOfLeftMostLeafOfRightSubtree($\theta_{i+1,i+d}$) $-i$
9 **for** k = StartIndexOffset to StopIndexOffset **do**
10 $Bi\Psi_{i,i+d} = Bi\Psi_{i,i+d} \bigcup \wedge\{\theta_{i,i+k} \cup \theta_{i+k+1,i+d}\}$
11 $\theta_{i,i+d}$ = ChooseBest($\theta_{i,i+d}, \wedge\{\theta_{i,i+k} \cup \theta_{i+k+1,i+d}\}$)
12 Result = $\theta_{1,n}$
end

The class of ST-optimal trees is the most general characterization of tree cost functions for which optimal alphabetic binary trees can be generated in $O(n^3)$. Likewise, the class of ST-optimal trees that satisfy the monotonicity principle is the most general characterization of tree cost functions for which optimal alphabetic binary trees can be generated in $O(n^2)$. Hu et al. [15] have proposed a generalization of tree cost functions called *regular* functions for which their $O(nlogn)$ algorithm produces optimal results. It is noteworthy that all trees constructed using a regular tree cost function are also ST-optimal. Since the algorithm they used is based on dynamic programming, i.e., it assumes that optimal alphabetic subtrees are sufficient to generate optimal alphabetic trees, it is natural that all regular functions should be subtree optimal.

We are not aware of any previous work to characterize tree cost functions that satisfy the monotonicity principle. However, we conjecture that all regular functions satisfy the monotonicity principle, in which case our run time of $O(n^2)$ is not much worse than $O(nlogn)$ run time of Hu-Tucker algorithm for regular functions. However, our algorithm will run in $O(n^2)$ for a much larger class of tree cost functions.

Furthermore, we can show that the most general sets of combining and tree cost functions known so far for which Huffman's algorithm generates optimal trees – namely, all quasi linear combining functions with Schur concave tree cost functions [20] – result in ST-optimal trees. It should be noted that these conditions are derived for non-alphabetic binary trees. Thus, our work encompasses not only the alphabetic trees, but also other non-alphabetic tree optimization problems. However, our proposed algorithm is only applicable to alphabetic trees. Unlike alphabetic trees, ST-optimality is not sufficient to guarantee polynomial runtime for optimal non-alphabetic binary tree generation.

3.4 SF-optimal trees

As described above, for non-binary ST-optimality of tree cost is not sufficient to guarantee polynomial runtime of alphabetic tree optimization. However, if $|\sum_{l=1}^{d-j} St\Psi_{l:i+j+1,i+d}|$ is restricted to a constant or even to a polynomial in $d-j$, algorithm 2.2 will run in polynomial time, e.g., if $|\sum_{l=1}^{d-j} St\Psi_{l:i+j+1,i+d}| \leq c$, runtime of algorithm 2.2 will be:

$$\sum_{d=1}^{n-1}[\sum_{i=1}^{n-d}(\sum_{j=0}^{d-1} c+d)]+n = \sum_{d=1}^{n-1}[(c+1)(n-d)d]+n = (c+1)n^3/6-(c-5)n/6 = O(n^3).$$

If $|\sum_{l=1}^{d-j} St\Psi_{l:i+j+1,i+d}| = 1$ (as is the case for binary trees as well as for non-binary trees when the stronger definition of SF-optimality is satisfied) as the left most branch will never have more than one set of siblings. Thus, inner loop in algorithm 2.2 is a constant time operation leading to a $O(n^3)$ run time for the algorithm.

According to the definition of SF-optimality, if the tree is SF-optimal, $|St\Psi_{l:i+j+1,i+d}| = 1; 1 \leq l \leq d-j$, giving $|\sum_{l=1}^{d-j} St\Psi_{l:i+j+1,i+d}| = d-j$. In this case algorithm 2.2 will run in time complexity given by:

$$\sum_{d=1}^{n-1}[\sum_{i=1}^{n-d}(\sum_{j=0}^{d-1}(d-j)+d)]+n = (n^4+6n^3-n^2-6n)/24 = O(n^4)$$

Intuitively, if a tree is SF-optimal, the cost contribution due to every subset of siblings is identifiable in terms of function H, allowing us to determine a priori if one k-tree subforest is better than another k-tree subforest on the same set of leaf nodes. Fortunately, conditions of SF-optimality are often satisfied, e.g., in case of Huffman's original additive combining function and tree cost function.

For generating degree-restricted optimal alphabetic trees, the above is modified to:

$$\sum_{d=1}^{n-1}[\sum_{i=1}^{n-d}(\sum_{j=0}^{d-1} min[d-j,t]+d)]+n \leq \sum_{d=1}^{n-1}[\sum_{i=1}^{n-d}(\sum_{j=0}^{d-1} t+d)]+n = (t+1)n^3/6-(t-5)n/6 = O(n^3t)$$

In this special case, however, our algorithm has reduced to the algorithm proposed by Vaishnavi et al. [39]. Application they were considering was t-ary alphabetic tree optimization with optimum average weighted search time. They also showed that for that application, no further reduction in run time is possible. Thus, their algorithm was derived for a specific tree cost function while we arrived at our algorithm from a more general algorithm. Hence, we can characterize tree cost functions for which we can achieve $O(n^3t)$ run time. An improved run time of $O(n^3 log t)$ has been achieved by Itai [19] and Gotlieb [11]. However, that algorithm is applicable to a restricted set of tree cost functions for which there is always an optimal tree with maximum allowable number of branches at the root.

It is intuitive to note that ST-optimality was sufficient for binary trees because the only proper subsets of a set of two children of an internal node contain only one subtree. As a matter of fact, even for SF-optimality it is sufficient to consider only proper subsets of the set of children of an internal node. For example, for trees with degree bounded by t, it is sufficient that SF-optimality is satisfied for up to $(t-1)$-tree subforests.

3.5 Summary

We have derived the number of alphabetic trees on n leaf nodes and provided a generic algorithm to produce alphabetic trees for any application. We showed that the exponential run time of this algorithm can be significantly improved when the tree is ST-optimal. We classified a set of tree cost functions that result in ST-optimal trees. ST-optimal tree cost function allows us to generate optimal alphabetic binary trees with the same time complexity as the best known algorithms. However, since our algorithm was derived using a top-down approach as opposed to deriving optimal alphabetic binary tree with respect to a specific tree cost function, we can

generalize and apply the algorithm to a larger class of tree cost functions. To derive optimal alphabetic non-binary trees in polynomial time, we proposed a further classification of tree cost functions, namely, SF-optimal trees. Again, we showed that in most cases, our algorithm reduces to the best known algorithm while being applicable for a much larger set of tree cost functions.

In the next section we consider applications of alphabetic trees in the field of logic synthesis. We apply the optimal alphabetic binary tree generation mechanism and optimal alphabetic non-binary tree generation to the fanout optimization problem. For additional details and another application to technology decomposition, please refer to [32, 38].

4 Alphabetic Fanout Optimization

Alphabetic fanout optimization problem may be stated as follows.

Problem 4.1 ALPHABETIC_FANOUT_OPTIMIZATION (ALPFANOUT)

- **Instance:**

  1. *A set of n sinks $(L_1, L_2, \ldots, L_n)$ in a given, fixed order with 2-dimensional weights $W_{L_i} = (r_{L_i}, \gamma_{L_i})$ where r_{L_i} = required time at L_i and γ_{L_i} = input load of sink L_i.*

  2. *A 2-dimensional combining function (f_{req}, f_{load}) which combines l nodes generating an internal node I_m with weight vector W_{I_m} where,*

$$\begin{aligned} r_{I_m} &= f_{req}(W_{child^1_{I_m}}, W_{child^2_{I_m}}, \ldots, W_{child^l_{I_m}}) \\ &= min_i(r_{child^i_{I_m}}) - \beta_{buf} \sum_i \gamma_{child^i_{I_m}} - \alpha_{buf} \qquad (22) \\ \gamma_{I_m} &= f_{load}(W_{child^1_{I_m}}, W_{child^2_{I_m}}, \ldots, W_{child^l_{I_m}}) \\ &= \gamma_{buf} \qquad (23) \end{aligned}$$

     $\beta_{buf}, \alpha_{buf}$ and γ_{buf} denote drive resistance, internal delay and input load of the buffer, respectively.

  3. *A tree cost function $C(T) = r_{I_m}$ where I_m is the root of the fanout tree T.*

- **Problem:** *Generate a tree T such that the cost (required time at the root) is maximum and the tree has no internal wire crossings*[5].

Notice that this formulation considers only maximization of the required time at the root of a fanout tree with fixed sink order while creating no internal wire crossings. Other cost factors such as the number of buffers used or the area of such buffers are not directly considered.

Combining function in ALPFANOUT corresponds to the *library* delay model. The problem can be easily modified to consider other delay models by restricting the values of different parameters. For example, if unit fanout delay model is used, $\alpha_{buf} = 0$, $\beta_{buf} = 1$ and $\gamma_{buf} = 1$. Under the library delay model, more than one buffers may be available with different $\beta_{buf}, \alpha_{buf}$ and γ_{buf}, adding an extra degree of complexity to the problem. To simplify the problem we use only one buffer type in our implementation. However, we discuss implications of allowing multiple buffers in section 4.2.

Lemma 4.1 *Alphabetic fanout trees are SF-optimal under the unit delay model and unit fanout delay model.*

Proof For the unit delay model, the proof follows from the constant value of f_{load} ($f_{load} = 0$). The optimal solution in this case is trivial, namely, a tree in which the root directly connects to all the sinks. Even under unit fanout delay model, value of f_{load} is a priori known for a D-rooted forests $\forall D$ ($f_{load} = c\ D$ = constant multiplied by number of outputs). Hence in either case, it is sufficient to keep the D-rooted forest with the maximum required time. The corresponding tree cost decomposition is shown in Figure 5. ■

Above lemmas imply that under the unit delay or unit fanout delay, alphabetic fanout optimization problem is optimally solvable in polynomial time. Even under library delay model the above lemmas hold if the loads are identical. However, with different loads or multiple buffers the alphabetic fanout trees are no longer SF-optimal.

Fortunately, even with different loads alphabetic fanout trees are still ST-optimal if there is only one buffer in the gate library.

Lemma 4.2 *Alphabetic fanout trees are ST-optimal the under library delay model if the library contains only one buffer/inverter.*

[5]The proposed method can be extended to limit the number of fanouts per buffer. However, in this paper, for sake of simplicity, we will assume that buffers have with no fanout limit.

Proof According to the definition of ST-optimality, a tree is subtree optimal if the tree cost is monotone in the tree cost of each of its subtrees. Since, we have only one internal node, all internal nodes have the same load. Thus, only required time at an internal node may change due to restructuring of the subtree structure at that internal node. As per the combining equation of alphabetic fanout optimization problem, increasing the required time at an internal node while maintaining the load at constant may never result in a decrease of the tree cost function (required time at the root). ■

As a result of above lemma and lemma 3.2, we can use algorithm 2.2 with $O(2^n)$ complexity for solving the ALPFANOUT problem optimally. Otherwise, we need to resort to algorithm 2.1. Let us denote the specialization of algorithm 2.2 or algorithm 2.1 (as the case maybe) for the ALPFANOUT problem as ALGALPFANOUT. However, before applying this algorithm, we analyze the ALPFANOUT problem in order to simplify the problem space and reduce the number of trees which need to be considered in order to generate an optimal alphabetic fanout tree. We refer to this reduced set of trees as **apropos trees** for alphabetic fanout trees, i.e., trees which are sufficient for the purpose of alphabetic fanout optimization.

The delay through a buffer is given by $\alpha_{buf} + \beta_{buf} \sum_{j \in FO_{buf}} \gamma_j$ where FO_{buf} denotes fanouts of the buffer . The wiring load is estimated dynamically based on the number of fanouts. This load can then be included in γ_j. Using this mechanism, the required time at an intermediate buffer k is given by $r_k = min_{j \in FO_k}(r_j) - \alpha_{buf} - \beta_{buf} \sum_{j \in FO_k} \gamma_j$.

The fanout tree generating rules given below do not undermine the optimality of the algorithm but improve its efficiency. Given a set of original sinks, these rules generate a modified set of sinks. An additional requirement for these rules to be valid is that the $\gamma_j \geq \gamma_{buf}$ for each sink L_j. This requirement is satisfied in most cases as input capacitance of the inverter is less than most other gates. Similar rules for the unit delay model were proposed in [3] for generation of t-ary minimax tree under the unit delay model.

Lemma 4.3 *Given n internal or sink nodes and if the input capacitance of the inverter is less than that of all sinks, application of the following rules does not undermine optimality of* ALGALPFANOUT.

Rule 1: *If $n \geq 1$ and $r_i \geq max(r_{i-1}, r_{i+1}), 1 \leq i \leq n$, make $r_i = max(r_{i-1}, r_{i+1})$.*

Rule 2: *If $max(r_i, r_{i+s+1}) \leq min(r_{i+1}, \ldots, r_{i+s}) - \alpha_{buf} - \beta_{buf} \sum_{j=1}^{s} \gamma_{i+j}$, replace the sequence $L_{i+1}, ..L_{i+s}$ of nodes by one node with required time $min(r_{i+1}, ...r_{i+s}) - \alpha_{buf} - \beta_{buf} \sum_{j=1}^{s} \gamma_{i+j}$.*

Figure 8: Illustration of Rule 1 and Rule 2

Proof Instead of providing an exhaustive case enumeration, we only give an outline of the proof. Each rule takes a set of sinks and produces a modified set of sinks which may be smaller.

Rule 1: For rule 1, r_i will either get JOINed with r_{i-1}, r_{i+1}, or an ancestor or r_{i-1} or r_{i+1}. In each case, the required time of r_i will get dominated by it's sibling, as long as $r_i \geq max(r_{i-1}, r_{i+1})$

Rule 2: Consider an optimal alphabetic tree on leaf nodes 1 to n. Rule 2 can be proved by considering all subtrees on some proper subset of leaf nodes $i+1, i+2, \ldots, i+s$ with a sibling which is either r_i or r_{i+s}, or any ancestor of r_i or r_{i+s}. Each such subtree can be replaced by a subtree with load γ_{buf} and a required time given by $min(r_{i+1}, ...r_{i+s}) - \alpha_{buf} - \beta_{buf} \sum_{j=1}^{s} \gamma_{i+j}$, without changing the required time at the root. Finally, keeping only one of these subtrees, while discarding the rest can never result in a decrease in the required time the the root. This exactly corresponds to application on rule 2. Thus, application of rule 2 never undermines optimality of ALGALPFANOUT. ■

Application of these rules is illustrated in Figure 8 a). To simplify the presentation, we assume the unit fanout delay model, i.e., $\alpha_{buf} = 1, \beta_{buf} = 1$ and $\gamma_j = 1$ for all j. As shown in the figure let the required time at sinks be given by vector $\mathbf{r}$ = (10, 14, 15, 14, 8, 8, 14, 12). Rule 1 is applied on L_3, while rule 2 is applied on (L_2, L_3, L_4) and on (L_7, L_8), generating internal nodes I_1 and I_2, respectively. At this stage, we reach an *impasse* as neither of the rules can be applied. Instead, if we had used unit delay model, these rules would have generated optimal alphabetic fanout trees in O(n) time complexity without encountering any impasse. For the *unit fanout* or the *library* delay model, impasse is likely to occur, in which case we resolve the impasse by resorting to the tree generation part of ALGALPFANOUT. Application of these rules, however, has reduced the solution space by reducing the number of sinks from 8 to 5.

4.1 Handling Different Loads

Let us denote the set of apropos trees on leaf nodes i through j as $Ap\Psi_{i,j}$. As seen from equation (18), because of the tree splitting term, we must generate all subtrees in $Ap\Psi_{i,j}; \forall i, j \leq n$. This number can be reduced significantly using the following theorem.

Let R_T, L_T, req_T and $load_T$ denote minimum of the required times at immediate children of the root of tree T, cumulative load offered by immediate children of the root of tree T, required time at the root of the tree T, and the load at the root of the tree T, respectively. Using this notation, required time for a tree T generated by $\wedge(\{T_l\} \times (.\dot{-}.T_r))$ is given by:

$$req_T = min(req_{T_l}, R_{T_r}) - \beta_{buf}(load_{T_l} + L_{T_r}) - \alpha_{buf}$$

Definition 4.1 *Given two trees T' and T'' on the same leaf nodes,*

- *T' and T'' are* non-inferior *with respect to each other if $(R_{T'} - R_{T''}) > \beta_{buf}(L_{T'} - L_{T''}) > 0$.*
- *T' is* superior *to T'' if $(R_{T'} - R_{T''}) > 0 \geq \beta_{buf}(L_{T'} - L_{T''})$.*
- *T' is* inferior *to T'' if $\beta_{buf}(L_{T'} - L_{T''}) \geq (R_{T'} - R_{T''}) > 0$.*

Theorem 4.4 *For fanout optimization, when generating the set of apropos trees on sinks j through m, it is sufficient to maintain a set of trees which are non-inferior with respect to each other, but superior to all other trees.*

Proof As per the notation proposed earlier, the set of apropos trees on sinks $j+1$ through m is denoted by $Ap\Psi_{j+1,m}$ and the set of all alphabetic trees on sinks $j+1$ though m is denoted by $\Psi_{j+1,m}$. When we split a tree $T_r \in \Psi_{j+1,m}$, it generates a forest F_r with minimum of the required time at the roots given by R_{T_r} and sum of loads at the roots given by L_{T_r}. We need to show that apropos trees are sufficient to generate optimal alphabetic fanout tree in any tree structure.

According to tree generation procedure we have to JOIN F_r with the best tree $T_l = \theta_{i,j}$, while generating a tree $T \in \Psi_{i,m}$. In this case, the sibling of F_r consist only of T_l. The required time at the root of the tree T is given by

$$req_T = min(req_{T_l}, R_{T_r}) - \beta_{buf}(load_{T_l} + L_{T_r}) \tag{24}$$

Let us consider any two trees $T'_r, T''_r \in \Psi_{j+1,m}$. Without loss of generality, we assume $R_{T'_r} \geq R_{T''_r}$. From the definition 4.1, T'_r is either superior, inferior or non-inferior with respect to T''_r. We will now show that if T'_r is superior or inferior with respect to T''_r, we need to consider only one of the two.

T'_r is superior to T''_r: Given $(R_{T'_r} - R_{T''_r}) > 0 \geq \beta_{buf}(L_{T'_r} - L_{T''_r})$, we need to prove the following:

$$min(req_{T_l}, R_{T'_r}) - \beta_{buf}(load_{T_l} + L_{T'_r}) > min(req_{T_l}, R_{T''_r}) - \beta_{buf}(load_{T_l} + L_{T''_r})$$

This is true from the definition of a superior tree irrespective of the value of $req_{T_l}, load_{T_l}$ and β_{buf}.

T'_r is inferior to T''_r: Given $\beta_{buf}(L_{T'} - L_{T''}) \geq (R_{T'} - R_{T''}) > 0$, we need to prove the following:

$$min(req_{T_l}, R_{T'_r}) - \beta_{buf}(load_{T_l} + L_{T'_r}) < min(req_{T_l}, R_{T''_r}) - \beta_{buf}(load_{T_l} + L_{T''_r}) \quad (25)$$

From equation (25) we obtain the following.

$$min(req_{T_l}, R_{T'_r}) - min(req_{T_l}, R_{T''_r}) < \beta_{buf}(L_{T'_r} - L_{T''_r})$$

From the definition (4.1), this is true for each of the three cases, $R_{T''_r} \leq R_{T'_r} \leq R_{T_l}$, $R_{T''_r} \leq R_{T_l} \leq R_{T'_r}$, and $R_{T_l} \leq R_{T''_r} \leq R_{T'_r}$.

Thus, when the siblings of F_r consist only of a single tree, i.e., T_l, only those trees from $\Psi_{j,m}$ need to be considered which are non-inferior with respect to each other.

When the sibling of F_r consist of more than one tree, the above can be proved similarly by considering req_{T_l} to be the minimum of the remaining siblings and $load_{T_l}$ to be the total of their load. ■

Theorem 4.4 enables us to reduce the number of apropos trees for ALPFANOUT by only generating the set of non-inferior trees $Ni\Psi_{i,m}$. As we generate each tree structure, we compare it with the trees in the current $Ni\Psi_{i,m}$, deleting inferior trees from $Ni\Psi_{i,m}$ in the process. We introduce a rule which will exploit this fact.

Rule 3: During tree generation on sinks i through m, current tree T is added to $Ni\Psi_{i,m}$ only if T is non-inferior to all the trees in $Ni\Psi_{i,m}$. If T is superior to some trees currently in $Ni\Psi_{i,m}$, we remove those trees from $Ni\Psi_{i,m}$ before adding T.

The correctness of rule 3 follows from theorem 4.4.

4.2 Handling Multiple buffers

If the library of gates contains multiple buffers/inverters, ST-optimality of alphabetic fanout optimization is not guaranteed. Let us consider subtrees generated on leaf nodes $i, \ldots, i+d$. For each buffer/inverter in the library we could generate the best alphabetic tree on these leaf nodes with the corresponding buffer/inverter being the root of the tree. Let us denote by $\mathcal{B}$ the set of buffers in the gate library. Let the gate driving the root of the fanout tree be G. So far, it was assumed that the root of the tree is also the same buffer. Here, along with allowing multiple buffers, we also allow a sink to be driven directly by the library gate corresponding to the function.

We extend the definition of superior and non-inferior trees between pairs of 1-tree forests.

Definition 4.2 *Given two trees on T' and T'' on same leaf nodes,*

- *T' and T'' are* non-inferior *with respect to each other if $(req_{T'} - req_{T''}) > \beta_{buf}(load_{T'} - load_{T''}) > 0$ or if $(R_{T'} - R_{T''}) > \beta_{buf'} L_{T'} - \beta_{buf''} L_{T''} > 0$.*
- *T' is* superior *to T'' if $(req_{T'} - req_{T''}) > 0 \geq \beta_{buf}(load_{T'} - load_{T''})$ or if $(R_{T'} - R_{T''}) > 0 \geq \beta_{buf'} L_{T'} - \beta_{buf''} L_{T''}$.*
- *T' is* inferior *to T'' if $\beta_{buf}(load_{T'} - load_{T''}) \geq (req_{T'} - req_{T''}) > 0$ or if $\beta_{buf'} L_{T'} - \beta_{buf''} L_{T''} \geq (R_{T'} - R_{T''}) > 0$.*

$\forall \;\; buf, buf', buf'' \in \mathcal{B} \cup \{G\}$.

In context of definition 4.2, theorem 4.4 can be modified and a corresponding rule 3 can be obtained. Since these are rather simple modifications, we do not discuss them in detail. However, a direct implication of allowing multiple buffers is that, for each subset of leaf nodes, instead of maintaining only one subtree, we may have to maintain as many subtrees as the number of buffers in the library. In addition, as per definition 4.2, the set of non-inferior tree may become large due to multiple subtrees that must be considered and maintained for one set of leaf nodes. However, since number of buffers is usually small (4-10 buffers for most libraries) and since all the inferior subtrees are filtered out during tree generation, the run times do not grow substantially because of considering multiple buffers.

4.3 Implementation and Experimental Results

The algorithm AlpFanout_Alg was implemented in the *SIS* environment. Mapped networks were optimized with AlpFanout_Alg after deriving an order on the fanout of each node using the ordering mechanism specified.

Algorithm 4.1 AlpFanout_Alg (Γ, Θ, Ω)
Γ *is an optimized mapped Boolean network*
Θ *is a vector of required times*
Ω *is the ordering mechanism*
begin
  for each node $n \in \Gamma$ (in preorder) **do**
    L = Order_sinks (Γ, n, Ω)
    L' = Reduce_sinks (L, Θ)
    η = Generate_best_AlpFanout_tree (n, L')
    update_network (Γ, n, η)
  end
end

Reduce_sinks reduces the number of sinks using *rule 1* and *rule 2*. Given a set of ordered sinks, Generate_best_AlpFanout_tree returns an optimal fanout tree on the ordered sinks L' using *rule 3* and the apropos tree generation equation (18). Application of *rule 3* during tree generation significantly improves efficiency of AlpFanout_Alg. This is illustrated in Figure 9 as described next.

Continuing with the previous example, we want to generate the optimal fanout trees on the modified set of sinks (L_1, I_1, L_5, L_7, I_2). From these 5 sinks, using the tree generation mechanism, we generate all alphabetic trees on every subset of ordered sinks of size 2 to 5. According to *rule 3*, every tree in the list of current apropos tree should be *non-inferior* to all others. Inferior trees are dropped from the current list of trees and are excluded from further consideration.

For this particular example, from equation (8), there are 4279 alphabetic fanout tree structures. Due to the monotone tree cost function, the number of apropos trees (i.e., trees that must be considered to find the optimal solution) is 127. Use of rules 1 and 2 reduces the number of sinks to 5, hence lowering the number of apropos trees to 31. Rule 3 eventually reduces

Figure 9: Illustration of AlpFanout_Alg

the total number of apropos trees on final set of sinks to 9. Note that rule 3 also reduces the number of apropos trees during the generation of subtrees. Since, on average, number of sinks for fanout optimization ranges between 3-6 sinks, in spite of being exponential in the worst case ALGALPFANOUT is quite fast in practice. This also shows how a detailed analysis of an exponential algorithm can lead to better runtimes of the optimal algorithm, alleviating of any need for non-optimal heuristic procedures.

Now we describe how we handled different polarities of the sinks and the mechanism to derive the order of sinks. Previous algorithms considered sinks with differing polarities separately. However, to maintain alphabetic order on the sinks irrespective of their polarities, we used the following mechanism. For every set of the sinks, we generated two fanout trees: one with positive polarity at the root and the other with negative polarity at the root. Apart from this, during every JOIN operation, we only joined the subtrees with identical polarities.

The ordering on the sinks was derived using different mechanisms[6]. The PLACE_ORDER imposes a sink ordering based on the sink positions derived from a global placement solution for the Boolean network. The idea is that since the placement solution captures the connectivity structure of the network and the addition of fanout tree does not modify the network structure to a great extent, we can rely on this placement solution for estimating the relative positions of the sinks after fanout optimization and placement. By using the placement information, due to the non-crossing property of the alphabetic trees, we are able to preserve the crossing number of the network during fanout optimization. The underlying intuition is that increased planarity due to reduced crossing number would improve the circuit routing.

The REQUIRED_ORDER generates a sink ordering based on the required times of the sinks. This option has been adopted by other researchers in the field. However, our fanout optimization algorithm is provably better than other algorithms as shown in the previous sections. The rationale behind this ordering is that, in general, it is desirable (from the performance point of view) to put sinks with similar required times at the same depth in the fanout tree.

Table 1 compares ALPFANOUT with SISFANOUT [37] for all multi-level logic benchmarks recommended in [41]. All circuits were first optimized in SIS using area optimizing script. The

[6]Depending on the secondary objective of fanout optimization (routing congestion, power efficiency, etc), other ordering mechanisms can be proposed and implemented. A strong point about our work is that it allows various optimization criteria to be considered during the fanout optimization with little or no degradation of the circuit performance.

circuit	SisFanout		AlpFanout			
			Req		Place	
	Area 10^3	Delay ns	Area 10^3	Delay ns	Area 10^3	Delay ns
C1355	2838	30.62	2566	30.56	2440	31.39
C1908	3803	45.02	3049	43.17	2855	46.16
C2670	5229	33.65	4805	30.59	4790	32.61
C3540	11531	65.99	10512	64.69	10600	66.71
C432	1307	40.31	1184	39.43	1008	39.32
C6288	27060	165.67	25939	167.72	21693	167.59
C7552	21209	80.91	22133	71.20	19090	67.06
9symml	1304	21.53	1162	21.59	1143	23.53
b9	593	11.23	561	10.94	559	11.56
dalu	11408	69.23	11497	70.84	10021	73.92
k2	11925	38.25	11570	36.82	10497	38.30
rot	5043	29.55	4699	29.51	4224	34.28
t481	6698	30.57	5930	29.56	5828	31.00
% Gain	100	100	92.9	97.3	86.3	101.6

Table 1: Comparison between fanout optimization in Sis and AlpFanout.

circuits were mapped using the SIS mapper in timing mode [36] and then optimized using two fanout optimization algorithms. After the fanout optimization the circuits were placed using GORDIAN [22] and routed using TimberWolf global router [25] and YACR2 detailed router [33].

The first two columns give results generated by SISFANOUT. SISFANOUT tries a number of fanout optimization algorithms (LT_trees, Two_Level, Balanced, etc) at each node and picks the best fanout solution. The columns denoted by REQ corresponds to the ALPFANOUT results with REQUIRED_ORDER. Last two columns corresponds to the PLACE_ORDER option of ALPFANOUT. For PLACE_ORDER the circuits were placed using GORDIAN and sink orders were derived from this placement.

As can be seen, we do better then SISFANOUT in area for all cases. The best performance results are obtained with the REQUIRED_ORDER, and the best routing (smallest chip area) is obtained with the PLACE_ORDER. Overall the PLACE_ORDER saves 14% chip area as compared to SISFANOUT without a significant performance degradation. Fanout trees generated by REQUIRED_ORDER are about 4% faster than PLACE_ORDER but at the cost of 6% increase in chip area.

ALPFANOUT runtimes are quite comparable with those of the SISFANOUT. On a Sun Sparc Station 2, for $C1355$, $C1908$, $C2670$, $C3540$, $C432$, $C6288$ and $C7552$, ALPFANOUT took 171.0, 169.4, 796.5, 595.7, 69.2, 1826.5, and 2634.0 seconds respectively, versus the SISFANOUT time which were 181.5, 215.8, 807.3, 639.1, 81.4, 2099.2 and 1931.4 seconds respectively.

5 Concluding Remarks

In this paper, a generic alphabetic tree generation procedure was introduced and the number of alphabetic trees on n leaves was computed. We introduced the concept of ST-optimal and SF-optimal trees which allow us to significantly reduce the solution space for generating optimal alphabetic trees. Using this concept we showed that optimal alphabetic binary trees on n leaf nodes can be generated in $O(n^3)$ time complexity for ST-optimal tree costs.

For fanout optimization we proposed an efficient fanout optimization algorithm which improves circuit performance while honoring an order restriction on the sinks. This ordering is derived based on an early global placement, analysis of the network structure, or required time constraints at the primary sinks. We also proposed a set of rules that reduce size of the solution space while maintaining the optimality where we also introduce the notion of non-inferior set

of fanout trees and use this to obtain further reduction in the solution space. For alphabetic fanout, optimization we obtained 14% improvement in chip area as compared to SISFANOUT without a significant performance degradation.

The improvements in routing area and the delay clearly indicate that routing issues should be considered at earlier stages of logic synthesis and that such integration could significantly improve the performance and chip area. Our results motivate us to apply the routing driven approach to the technology independent phase of logic synthesis, specifically, logic decomposition and kernelization procedures. We hope that this theory will provide an effective way of incorporating routing issues (e.g., wire crossing, congestion) into logic synthesis.

References

[1] C. L. Berman and J. L. Carter. The fanout problem: From theory to practice. In C. L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the 1989 Decennial Caltech Conference*, pages 69–99. MIT Press, May 1989.

[2] A. Cayley. On the analytical forms called trees. second part. In *The Collected Mathematical Papers of Arthur Cayley: Volume 4*, pages 112–115. Cambridge University Press: Reprint–Johnson Reprint Corporation, New York, New York, 1891. From The Philosophical Magazine, vol. XVIII. (1859),374-378.

[3] D. Coppersmith, M. M. Klawe, and N. J. Pippenger. Alphabetic minimax trees of degree at most t. *SIAM Journal of Computing*, 15:189–192, 1986.

[4] Shimon Even. *Graph Algorithms*. Computer Science press, Rockville, Maryland, first edition, 1979.

[5] R. G. Gallager. Variations on a theme by Huffman. *IEEE Transactions on Information Theory*, IT-24(6):668–674, November 1978.

[6] M. R. Garey. Optimal binary identification procedures. *SIAM Journal of Applied Mathematics*, 23(2):173–186, September 1972.

[7] M. R. Garey. Optimal binary search trees with restricted maximal depth. *SIAM Journal of Computing*, 3(2):101–110, June 1974.

[8] E. N. Gilbert and E. F. Moore. Variable-length binary encoding. *Bell Systems Technical Journal*, 38:933–968, 1959.

[9] C. R. Glassey and R. M. Karp. On the optimality of Huffman trees. *SIAM Journal of Applied Mathematics*, 31(2):368–378, September 1976.

[10] M. C. Golumbic. Combinatorial merging. *IEEE Transactions on Computers*, 25(11):514–526, 1976.

[11] L. Gotlieb. Optimal multi-way search trees. *SIAM Journal of Computing*, 10:422–433, 1981.

[12] L. Gotlieb and D. Wood. The construction of optimal multiway search trees and the monotonicity principle. *International J. Computer Maths*, 9:17–24, 1981.

[13] H. J. Hoover, M. M. Klawe, and N. J. Pippenger. Bounding fanout in logical networks. *Journal of the Association for Computing Machinery*, 31(1):13–18, January 1984.

[14] Y. Horibe. An improved bound for weight-balanced tree. *Information and Control*, 34:148–151, 1977.

[15] T. C. Hu, D. J. Kleitman, and J. K. Tamaki. Binary trees optimum under various criteria. *SIAM Journal of Applied Mathematics*, 37(2):246–256, October 1979.

[16] T. C. Hu and M. T. Shing. Computation of matrix chain products. part I. *SIAM Journal of Computing*, 11(2):362–373, May 1982.

[17] T. C. Hu and A. C. Tucker. Optimal computer search trees and varaible-length alphabetical codes. *SIAM Journal of Applied Mathematics*, 21(4):514–532, December 1971.

[18] D. A. Huffman. A method for the construction of minimum redundancy codes. In *Proceedings of the IRE*, volume 40, pages 1098–1101, September 1952.

[19] A. Itai. Optimal alphabetic trees. *SIAM Journal of Computing*, 5(1):9–18, 1976.

[20] D. S. Parker Jr. Conditions for optimality of the Huffman algorithm. *SIAM Journal of Computing*, 9(3):470–489, 1980.

[21] D. G. Kirkpatrick and M. M. Klawe. Alphabetic minimax trees. *SIAM Journal of Computing*, 14(3):514–526, 1985.

[22] J. M. Kleinhans, G. Sigl, F. M. Johannes, and K. J. Antreich. GORDIAN: VLSI placement by quadratic programming and slicing optimization. *IEEE Transactions on Computer-Aided Design*, CAD-10:356–365, March 1991.

[23] D. Knuth. *The Art of Computer Programming*. Addison-Wesley, 1973.

[24] D. E. Knuth. Optimum binary search trees. *Acta Informatica*, 1:14–25, 1971.

[25] K. W. Lee and C. Sechen. A new global router for row-based layout. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 180–183, November 1988.

[26] T. Lengauer. *Combinatorial algorithms for integrated circuit layout*. Computer Science. Wiley-Teubner, 1990.

[27] Y. Lirov and O. Yue. Circuit pack troubleshooting via semantic control I: Goal selection. In *Proceedings of the International Workshop on Artificial Intelligence for Industrial Applications*, pages 118–122, 1988.

[28] T. Motzkin. Relations between hypersurface cross ratios, and a combinatorial formula for partitions of a polygon, for permanent preponderance, and for non-associative products. *Bulletin of American Mathematical Society*, 54:352–360, 1948.

[29] N. Nakatsu. Bounds on the redundancy of binary alphabetic codes. *IEEE Transactions on Information Theory*, 37(4):1225–1229, July 1991.

[30] K. R. Pattipati and M. G. Alexandridis. Application of heuristic search and information theory to sequential fault diagnosis. *IEEE Transactions on Systems, Man, and Cybernetics*, 40(4):872–887, 1990.

[31] M. Pedram and N. Bhat. Layout driven technology mapping. In *Proceedings of the 28th Design Automation Conference*, pages 99–105, June 1991.

[32] M. Pedram and H. Vaishnav. Technology decomposition using optimal alphabetic trees. In *Proceedings of the European Conf. on Design Automation*, pages 573–577, March 1993.

[33] J. Reed, A. Sangiovanni-Vincentelli, and M. Santamauro. A new symbolic channel router: YACR2. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 208–219, July 1985.

[34] J. Riordan. *Combinatorial Identities*. Robert E. Krieger Publishing Company, Huntington, New York, revised edition, 1979.

[35] K. J. Singh and A. Sangiovanni-Vincentelli. A heuristic algorithm for the fanout problem. In *Proceedings of the 27th Design Automation Conference*, pages 357–360, June 1990.

[36] H. J. Touati, C. W. Moon, R. K. Brayton, and A. Wang. Performance-oriented technology mapping. In *Proceedings of the Sixth M.I.T. Conference on Advanced Research in VLSI*, pages 79–97, April 1990.

[37] Herve Touati. *Performance Oriented Technology Mapping*. PhD thesis, University of California, Berkeley, 1990.

[38] H. Vaishnav and M. Pedram. Routability-driven fanout optimization. In *Proceedings of the 30th Design Automation Conference*, pages 230–236, June 1993.

[39] V. K. Vaishnavi, H. P. Kriegel, and D. Wood. Optimum multiway search trees. *Acta Informatica*, 14:119–133, 1980.

[40] R. L. Wessner. Optimal alphabetic search trees with restricted maximal height. *Information Processing Letters*, 4(4):90–94, 1981.

[41] S. Yang. Logic synthesis and optimization benchmarks user guide: Version 3.0. , Jan 1991.

[42] R. W. Yeung. Alphabetic codes revisited. *IEEE Transactions on Information Theory*, 37(3):564–572, May 1991.