Optimal Combinational Multi-Level Logic Synthesis

by

Elizabeth Ann Ernst

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2009

Doctoral Committee:

Professor Karem A. Sakallah, Chair
Emeritus Professor Edward S. Davidson
Professor John P. Hayes
Professor Stephane Lafortune
Professor Marios C. Papaefthymiou

Acknowledgements

First, I would like to thank my advisor Professor Karem Sakallah without whom this work would not have been possible. His guidance in developing, executing and presenting this research has been an invaluable asset to me. I also thank my committee members Ed Davidson, John Hayes, Stephane Lafortune, and Marios Papaefthymiou for their thoughtful comments and suggestions. I would particularly like to thank Ed Davidson for the brainstorming sessions which produced many insights into this work. I am grateful for the opportunity to work with him.

I would like to acknowledge all the friends, class-mates, and office-mates I've had during my time at Michigan. Each person made my graduate student experience truly unique and enjoyable. I would like to especially thank Tara for our weekly lunches. I always enjoyed that one break during the long week to talk, laugh, and commiserate about both work and life.

And finally, I want to thank my family for all they have given me. To my parents Larry and Janis, for their love and unwavering support throughout my many years of schooling. And to my husband Dan for all he has endured during this journey (including reading this thesis more times than he cares to admit). His love, encouragement, and infinite patience were always present to help me along the way.

Table of Contents

List of Figures

Abstract

Within the field of automated logic design, the optimal synthesis of combinational logic has remained one of the most basic design objectives. However, the computational complexity of this optimization problem has limited the practical application of optimal synthesis for large circuits. Since much of the exact synthesis literature predates many advances in both computer hardware as well as reasoning and search techniques, it was our objective to revisit optimal synthesis. Through this investigation we hoped to complete optimal synthesis on more complex functions.

In this dissertation, we provide a general formulation of logic synthesis as an expanding search problem and describe BESS, an optimal multi-level branch-and-bound synthesis algorithm for combinational circuits. The formulation of synthesis as an expanding search problem provides insights into the difficulty of optimal synthesis. The generality of the formulation provides the flexibility in the options under which synthesis could be completed. Since BESS was created based on this formulation, it completes optimal synthesis under a variety of options while guaranteeing that an optimal network will be produced

In this dissertation, we provide a comprehensive evaluation of BESS. First we describe an extensive study of the search strategies for BESS, including both empirical and theoretical arguments which explain why these strategies are able to provide a more efficient search. Then through an analysis of the algorithm, we provide proofs of both completeness and convergence as well as an analysis of the search space.

Empirically, we discuss the findings yielded by BESS. We give a database of optimal circuits. Optimal implementations of all 2-, 3-, and 4-input functions are given, including for the first time the optimal implementation of the 4-input xor function. We then extend this database of know optimal circuits to include 4,745 5-input functions. We also provide optimal networks and cost formula for n-input functions for a variety of common functions based on an analysis of optimal network structures. Finally, we provide networks for larger function that are with-in a known distance from optimal by modifying the bounding technique in BESS. Networks with as many as 17 inputs and 16 outputs are completed.

Chapter 1

Introduction

1.1 Optimization in Automated Circuit Design

In circuit design, the translation from behavioral model to circuit layout is an arduous task for a human designer. For this reason, automatic synthesis was developed to take most of the burden off of the designer by automating many of the stages in the design process. Since its creation in the 1950s, automated synthesis has grown quickly into a widely relied-upon technique for computer engineering. As a direct result of Moore's law on the growth in circuit density, digital logic designs have become both more complex and vastly more common. Because of this, the demand for quick, simple, and efficient logic design automation tools has exploded into a multi-billion dollar market.

There are three stages in the typical automatic digital design process. In the first stage, behavioral synthesis, a high-level description of the logic system is translated to a register transfer level (RTL) implementation. At the RTL, the logic system's behavior is defined in terms of the transfer of data between registers and the logical operations performed on the data. From the RTL implementation, logic synthesis is performed producing a gate-level description of the circuit using a pre-defined set of logic gates. Next, placement, technology mapping and routing are completed, producing a physical layout of the circuit. This layout is a representation of the integrated circuit which corresponds to the semiconductor layers that make up the components of the circuit.

The automatic synthesis of a logic system that meets the specifications set by the designer is a difficult but feasible task. However, there are many circuit layouts for a given behavioral model, some of which may be more desirable to the designer. Thus, an optimal circuit layout is sought. This additional optimality constraint adds a layer of complexity to the automated design process. Since circuit design is divided into stages, the optimality of the final result imposes an optimality constraint on each stage of the design process. In the work presented here, we focus on optimization within the logic synthesis portion of the overall automated design. Logic synthesis is often further divided into sequential and combinational synthesis. Sequential synthesis focuses on the registers in the RTL implementation while combinational

synthesis focuses on the logic operations that exist between the registers. We consider only combinational synthesis in this work.

Specifically, our work focuses on the task of optimal combinational logic synthesis: the translation of the combinational portion of a logic system to a netlist of gates while optimizing this netlist according to a set of criteria. The criteria commonly used for combinational logic synthesis optimization includes one or more of the following: the area occupied by the logic gates and interconnect, the critical path delay of the longest path through the logic, the degree of testability of the circuit, and the power consumed by the logic gates.

Combinational synthesis is often divided into two categories based on a gate delay constraint that may be placed on the resulting netlist. In two-level synthesis, the critical path delay is restricted to two levels and the logic design is optimized based on the area occupied by the gates in the netlist. In multi-level synthesis, the gate depth restriction is removed. In this case, the optimal designs present trade-offs between area and delay of the circuit. The area of a circuit can often be reduced from the minimum value obtained in two-level synthesis by trading an increase in the gate delay for this savings.

The optimization problem of two-level logic synthesis is well-understood. Many powerful tools [Coudert 03][Rudell 87][Sapra 03][McGeer 93] exist for solving two-level minimizations and functions with hundreds of inputs and outputs can be completed optimally with these tools. However, two-level synthesis has limited use in most VLSI designs as these systems typically require multiple levels of logic.

The additional degree of freedom created by the removal of the delay restriction makes the optimization problem of multi-level synthesis much more complex than its two-level counterpart. The increased potential for reusing gates in a netlist produces more freedom in the possible solution structures. This increases the size of the solution space over the two-level optimization problem and, as a result, multi-level netlists are more difficult to synthesize optimally.

Due to the difficulty of the optimization problem and yet the need for automated solutions to this problem, most of the research in the area of multi-level synthesis has been devoted to heuristics that meet constraints on selected criteria while attempting to minimize the other criteria as much as possible. The goal of these heuristics is to find a near-optimal netlist in a reasonable amount of time for functions with a large number of inputs and outputs. However, these heuristics forfeit the optimality of the solution.

In spite of its difficulties, the problem of optimal multi-level combinational synthesis (or exact synthesis) has continued to intrigue researchers. The optimization of combinational logic synthesis still remains one of the most basic objectives of conventional switching theory. The "good enough" nature of the results produced by the heuristics has led the focus of multi-level synthesis away from the problem of optimality. The purpose of this research is to return some focus back to the original problem.

The long history of work in the area of optimal synthesis provides the basis for our research. This previous work has allowed us to better understand the difficulty of optimal synthesis. Using this

knowledge, we improve upon these earlier methods by employing modern reasoning techniques such as exploiting functional symmetry and applying learning techniques and by employing more effective heuristics. Further improvements due to advancements in computer and circuit technology are also observed. Ultimately, we are led to a more efficient algorithm for optimally decomposing larger and more complex sets of functions.

1.2 Methods for Optimal Synthesis

The investigation of optimal synthesis has been quite limited compared to the research invested in heuristic methods for multi-level synthesis. However, there have been several different approaches taken towards completing synthesis optimally. The methods for optimal synthesis can be divided into categories based on these different approaches.

The first such category takes a functional approach by using functional decomposition. Synthesis is completed by performing repetitive decompositions until all functions of the decomposition are contained in the specified set of building blocks. Optimal synthesis is performed by searching through all possible ways of performing these functional decompositions. In the early works on logic synthesis this was the method used. Originally the methods were completed using decomposition charts [Ashenhurst 59][Curtis 61] but later more compact forms for manipulating the decompositions of Boolean functions were found [Karp 61][Roth 60][Roth 62]. These methods are well suited for finding a network implementation for the given function quickly. Once an initial network has been completed, the remainder of the time is spent searching through the solution space either improving upon the initial network or proving that the found network is optimal. Because of the functional decompositions employed by these algorithms, it is often difficult to switch among building block sets which are used to build the networks. If a new set of building blocks is desired, a complete reworking of the decomposition rules is required.

A structural approach through network enumeration forms the second category of methods used to complete optimal synthesis. Network enumeration is based on the fact that all possible networks with n gates can be enumerated. Using such an enumeration, the space of n gate networks can be searched for one that generates the desired function. If this search process is repeated, beginning with $n = 1$, for an increasing number of gates then the first network found during the search is an optimal network with respect to a cost function based on the number of gates in the network. The enumeration of these networks can be done either explicitly [Hellerman 63][Smith 65][Drechsler 98] or implicitly [Muroga 72]. When performed explicitly, each network in the enumeration is generated and the resulting function checked against the desired function. When performed implicitly, the space of all possible networks are represented by a single structure. A constraint problem is obtained from this structure such that a solution which satisfies all the constraints gives the network structure which generates the desired function. The simplicity of the enumeration methods allow for an easy change of the logic gates used as building blocks. Since an

optimal cost network is not found until all possible networks with smaller cost have been searched, the majority of the work done using these methods is completed before a network for the function is found.

The third approach to optimal synthesis combines techniques from both the functional and structural methods of the previous two approaches. These methods [Davidson 68b][Nakagawa 89] perform a search over the space of all networks which implement the function. The search is performed by dividing the set of possible solution networks at a branching step into smaller sets until a set containing a single network is found. The methods used to divide the search are based on both structural and functional properties of the networks. These methods are able to maintain some of the simplicity of the enumeration methods while incorporating the searching methods from the functional decomposition methods. The search employed by these methods allows an initial network to be found relatively quickly for any function. The bulk of the work is then spent on improving the network and proving the optimality of the final network.

1.3 Contributions and Findings in Optimal Synthesis

In this thesis we provide a theoretical and experimental evaluation of optimal synthesis. Using modern reasoning and search techniques as well as advancements in computer technology we complete optimal synthesis on more complex functions. Over the course of this work we provide the following contributions and findings:

1. A precise formulation of optimal synthesis as an expanding search problem.

   We provide a general formulation of multi-level logic synthesis as a dynamic search problem. The formulation of synthesis as a search problem provides insight into the difficulty of optimal synthesis observed by previous researchers. Unlike many typical search algorithms, the entire set of decision variables is not known before the search begins. Thus, as the network grows during synthesis, the search space expands as well. The generality of this formulation allows us to easily adapt our method to provide flexibility in the options under which synthesis is completed.

2. The development and analysis of an optimal synthesis algorithm BESS.

   Using the formulation provided, we created an optimal synthesis algorithm, BESS (Branch-and-bound Exact Synthesis System). Since this algorithm is based on the general formulation of synthesis through search, it can easily handle a variety of synthesis options while guaranteeing the optimality of the result. Due to the branch-and-bound method used to complete the search, the algorithm also allows for the possibility of relaxing the optimality constraint so that networks for larger functions can be completed with near-optimal cost.

   When creating this algorithm, we performed an extensive study of various search strategies including branching heuristics, the propagation of both functional and structural implications, and the removal of redundant search through pruning rules. The comprehensive evaluation

described here includes both empirical and theoretical arguments to explain why these strategies are able to provide a more efficient search.

We also provide an analysis of this algorithm. Our proof of completeness shows that BESS will explore all possible implementations of a Boolean function guaranteeing that the optimal implementation will be found. In providing a proof of convergence we discuss three requirements that must be added to the initial path of the search in order to guarantee that an initial network will be completed and that the search will eventually complete. Finally, we provide an analysis of the search space which shows, for the first time, the true difficulty of optimal synthesis. Using a conventional worst-case analysis of the width and height of the search tree a greater than double exponential upper bound on the size of search space is given based on the number of inputs. Later, using empirical results we obtain a more reasonable estimate of the search space based on the number of gates in the optimal network.

3. A database of optimal circuits for an extensive collection of functions

   Using BESS we completed a database of known optimal networks. Previous work had provided optimal circuits for all 2-, 3-, and all but one 4-input functions. We first confirmed the optimality of these existing results and then extended the database to include the additional unknown 4-input function (the 4-input xor function) as well as 4,745 new 5-input functions.

   In creating this database, we first observed the combinatorical difficulty of optimal synthesis. Despite the improvements made to the algorithm and the advancements in technology, the problem of optimal synthesis remains difficult for functions requiring more than 16 gates. While small, this is an advancement over previous results where networks could only be completed with a maximum of 12 gates.

4. Optimal networks and cost formula for n-input functions for a variety of common functions

   By analyzing the empirical evidence provided by BESS on a variety of common functions we were able to identify patterns within their networks. Using these patterns we provide optimal implementations and cost formula for the n-input functions from these classes and prove them to be optimal for half of the classes.

5. Networks for larger functions that are a known-distance from optimal

   By modifying the bounding technique in BESS we obtain near-optimal networks for larger functions. These networks are a known distance from optimal, yet are significantly larger than the results obtained by optimal synthesis. Here we provide near-optimal networks for functions with as many as 17 inputs and 16 outputs.

1.4 Thesis Overview

Over the course of this thesis we will provide the details of our work outlined in the previous section. We begin in Chapter 2 by giving the framework of optimal synthesis. This introduces the definitions and the notation we use to describe the properties of Boolean functions as well as properties and terminology that are needed for discussing circuits and their representations. In the third chapter, we provide the theoretical framework for our optimal synthesis algorithm. In addition to a description of the basic algorithm, we include a discussion of the solution space that the algorithm must search in order to produce an optimal circuit and also describe improvements that can be made to the algorithm which will help to reduce search space while still maintaining the optimality of the resulting circuit.

Improvements and variations of BESS are the focus of Chapters 4 through 6. In Chapter 4 we present the experimental evaluation of the algorithm. In addition to producing optimal results, we also build upon the theoretical analysis given in Chapter 3. An analysis of various classes of functions based on their optimal circuits are also given. Chapter 5 provides the results obtained from variations on the basic algorithm. While still producing optimal circuits, these variations include changes to the fan-in and fan-out restrictions, to the building block set, to the level restrictions, and to the cost function. A comparison of these variation results with the original results provides a way to more fully understand the problem of optimal synthesis. The results from these variations are also used to compare BESS to other methods for completing optimal synthesis. Finally, in Chapter 6, we present several variations on the algorithm which produce near-optimal results. We evaluate methods for producing near-optimal results and finally use these results to evaluate an existing heuristic methods.

We conclude this thesis with Chapter 7. There, we discuss several possible directions for future research work including how to expand BESS for new variations, how this algorithm can be used to evaluate heuristic methods, or used in conjunction with heuristic methods. Finally, we conclude with an overview of the contributions of the thesis.

Chapter 2

Definitions and Notation

2.1 Exact Synthesis Overview

Combinational logic synthesis is the process of decomposing a set of Boolean functions into a design implementation in terms of a fixed set of logic gates. Exact synthesis performs logic synthesis while minimizing a cost function over all possible implementations. The specification for exact synthesis requires a set of Boolean functions for decomposition, a set of building blocks, and a cost function.

The set of building blocks given in the problem specification is the fixed set of logic gates that are used in the design implementation of the Boolean functions. This set of building blocks must be composed of a functionally complete set of logic gates. A set of gates is **functionally complete** if any Boolean function can be realized using only gates from the set [Wernick 42].

For example, the set of gates {AND, OR, NOT} is functionally complete. Any set of gates which contains this set as a subset would also be functionally complete. Other common complete sets of gates are the singleton sets {NAND} and {NOR}.

A cost function will also be needed as part of the problem specification for exact synthesis. This cost function must be a measurable attribute of a circuit. The values produced by the cost function must have a strict ordering so that the cost of two circuits can be compared and the circuits ordered based on this function. Such cost functions could include the number of gates in the circuit (active area of the circuit), the number of gates in the longest path of the circuit (depth of the circuit), or the number of literals needed for an algebraic expression representation of the circuit. In addition, weights could be associated with each gate in the circuit while the cost function would be the sum of the weights to each gate in the circuit. Finally the cost function could be based on some combination of the above choices.

2.2 Function Representation

Let $f(x_1,\ldots,x_n)$ be a partially-specified n-variable Boolean function. Such a function can be expressed by its on-, off-, and don't-care sets. Alternatively, the function can be expressed as an interval in the space of n-variable Boolean functions. The algorithm is based on the on- and off-set representation. The interval representation is used in derivations.

2.2.1 On-set/Off-set Representation

$f(x_1,\ldots,x_n)$ can be expressed in terms of three completely specified n-variable Boolean functions defined as follows:

$$\mathrm{ON}[f](x_1,\ldots,x_n)=\begin{cases}1 & \text{when } f(x_1,\ldots,x_n)=1\\ 0 & \text{otherwise}\end{cases} \tag{1}$$

$$\mathrm{OFF}[f](x_1,\ldots,x_n)=\begin{cases}1 & \text{when } f(x_1,\ldots,x_n)=0\\ 0 & \text{otherwise}\end{cases} \tag{2}$$

$$\mathrm{DC}[f](x_1,\ldots,x_n)=\begin{cases}1 & \text{when } \left(f(x_1,\ldots,x_n)\neq 0\right) \text{ and } \left(f(x_1,\ldots,x_n)\neq 1\right)\\ 0 & \text{otherwise}\end{cases} \tag{3}$$

For simplicity, in what follows, we will drop these functions' explicit dependence on $x_1,\ldots,x_n$. Thus, $\mathrm{ON}[f]$ should be implicitly understood to be a completely-specified n-variable Boolean function defined according to (1). These three functions form a partition of the n-dimensional Boolean space, i.e., they are pair-wise disjoint and their union covers the entire space. Thus, we only need to specify two of them to capture f. In particular, if we choose to express f in terms of $\mathrm{ON}[f]$ and $\mathrm{OFF}[f]$, then $\mathrm{DC}[f]$ can be determined from:

$$\mathrm{DC}[f]=\left(\mathrm{ON}[f]\vee\mathrm{OFF}[f]\right)'=\mathrm{ON}'[f]\wedge\mathrm{OFF}'[f] \tag{4}$$

When $\mathrm{DC}[f]$ is identically 0, then f is *completely specified.* Otherwise, f is *partially specified.* The disjointness of the on- and off-sets can be expressed by the constraint

$$\left(\mathrm{ON}[f]\wedge\mathrm{OFF}[f]\right)'=\mathrm{ON}'[f]\vee\mathrm{OFF}'[f] \tag{5}$$

which insures that $\mathrm{ON}[f]$ and $\mathrm{OFF}[f]$ cannot be simultaneously true (i.e. equal to 1 for the same minterm).

2.2.2 Function Interval Representation

$f(x_1,\ldots,x_n)$ can also be expressed in terms of two completely-specified n-variable Boolean functions that define an interval in the space of n-variable Boolean functions. Specifically,

$$f(x_1,\ldots,x_n)=\left[\mathrm{LB}[f],\mathrm{UB}[f]\right] \tag{6}$$

where the interval notation is a compact representation of the set of functions $\{f \mid \mathrm{LB}[f] \le f \le \mathrm{UB}[f]\}$ [Hachtel 96]. The two functions specifying the interval are referred to as the *lower-* and *upper-bounds* of the interval. For the interval to be non-empty, the following constraint must be satisfied:

$$(\mathrm{LB}[f] \le \mathrm{UB}[f]) = (\mathrm{LB}'[f] \vee \mathrm{UB}[f]) \tag{7}$$

2.2.3 Relations Among Two Representations

The functions used in these two representations are related as follows:

$$\begin{aligned} &\mathrm{ON}[f]=\mathrm{LB}[f] \\ &\mathrm{OFF}[f]=\mathrm{UB}'[f] \\ &\mathrm{DC}[f]=\mathrm{LB}'[f]\wedge\mathrm{UB}[f] \end{aligned} \tag{8}$$

$$\begin{aligned} &\mathrm{LB}[f]=\mathrm{ON}[f] \\ &\mathrm{UB}[f]=\mathrm{ON}[f]\vee\mathrm{DC}[f]=\mathrm{OFF}'[f] \end{aligned} \tag{9}$$

2.3 Network Model

The input to the synthesis problem is a set of p completely- or partially-specified n-variable Boolean functions:

n Primary inputs: $x_1,\ldots,x_n$

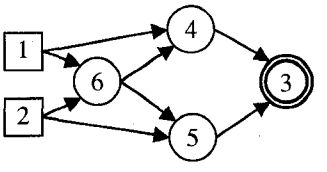
p Primary outputs: $\begin{cases} f_{n+1}(x_1,\ldots,x_n) \\ \cdots \\ f_{n+p}(x_1,\ldots,x_n) \end{cases}$ assumed to be completely- or partially-specified n-variable Boolean functions

In addition, a set of building blocks and a cost metric are also required. We will use NAND2 gates as the building block for the networks and the total number of gates in the network as the cost metric. The output of the synthesis process is a minimum-cost multi-level network of NAND2 gates. (A description of the algorithm with alternate building blocks and cost metrics will be given in Chapter 5). The network will be synthesized incrementally by adding NAND2 gates and by adding connections between those gates and

between the primary inputs and those gates. During the synthesis process, the evolving multi-level implementation will be represented by a DAG with q nodes labeled with integers from 1 to q:

- Nodes labeled $1,\ldots,n$ correspond to the primary inputs
- Nodes labeled $n+1,\ldots,n+p$ correspond to the primary output NAND2 gates
- Nodes labeled $n+p+1,\ldots,q$ correspond to the internal NAND2 gates

Primary output nodes and internal nodes will be referred to as gate nodes. A gate node will be denoted by a 3-tuple (i,j,k) where i is the label of the node itself, and j and k are the labels of the nodes which are its fan-ins. In other words, (i,j,k) specifies the existence of 3 nodes i, j, and k, and two edges (j,i) and (k,i). An unconnected input of a gate node will be indicated by 0. Thus $(i,j,0)$ specifies a node with one unconnected input, and $(i,0,0)$ specifies a node with two unconnected inputs. Figure 2.1 illustrates this notation for a small example network.



Primary Input Nodes: 1, 2

Primary Output Node: (3, 4, 5)

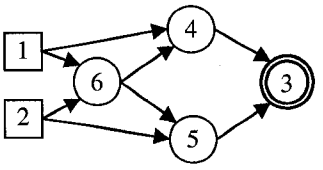
Internal Nodes: (4, 1, 6), (5, 6, 2), (6, 1, 2)

Figure 2.1: Network Notation Example

Each node, including primary input nodes, will have an associated n-variable Boolean function $f_i(x_1,\ldots,x_n)$.

- For primary input node i, $f_i(x_1,\ldots,x_n)=x_i$
- For gate node (i,j,k), $f_i(x_1,\ldots,x_n)=f_j'(x_1,\ldots,x_n)\vee f_k'(x_1,\ldots,x_n)$. If the gate node is a primary output node, its function is provided in the problem specification. The functions of internal nodes are derived as part of the synthesis process.
- For the pseudo-node 0, we define $f_0(x_1,\ldots,x_n)=[0,1]$. Note that $\mathrm{ON}[f_0]=\mathrm{OFF}[f_0]=0$.

During the synthesis process, node functions will generally be partially specified and represented by on- and off-sets. To simplify notation, we use ON_i and OFF_i, to denote $\mathrm{ON}[f_i]$ and $\mathrm{OFF}[f_i]$ respectively. Figure 2.2 illustrates the Boolean functions associated with the nodes of the network from Figure 2.1.



Node	ON_i	OFF_i
1	x_1	x_1'
2	x_2	x_2'
(3, 4, 5)	$(x_1 \wedge x_2') \vee (x_1' \wedge x_2)$	$(x_1 \wedge x_2) \vee (x_1' \wedge x_2')$
(4, 1, 6)	$x_1' \vee x_2$	$x_1 \wedge x_2'$
(5, 6, 2)	$x_1 \vee x_2'$	$x_1' \wedge x_2$
(6, 1, 2)	$x_1' \vee x_2'$	$x_1 \wedge x_2$

Figure 2.2: A Network and its Associated Boolean Functions: The functions associated with the output node (3,4,5) is provided as part of the network specification.

2.4 Basic NAND2 Relations

Let (i, j, k) be a gate node. The relations among the node's input and output functions can be expressed as follows:

$$\begin{aligned}[\mathrm{LB}_i, \mathrm{UB}_i] &= [\mathrm{LB}_j, \mathrm{UB}_j]' \vee [\mathrm{LB}_k, \mathrm{UB}_k]' \\ &= [\mathrm{UB}_j' \vee \mathrm{UB}_k', \mathrm{LB}_j' \vee \mathrm{LB}_k']\end{aligned} \tag{10}$$

These relations can be expressed equivalently in terms of the on- and off-set functions:

$$\begin{aligned}\mathrm{ON}_i &= \mathrm{OFF}_j \vee \mathrm{OFF}_k \\ \mathrm{OFF}_i &= \mathrm{ON}_j \wedge \mathrm{ON}_k\end{aligned} \tag{11}$$

Additionally, these relations along with (5) give us all the "implications" required to keep the inputs and output of the gate consistent when the on- and off-sets change during the synthesis process. Specifically:

- Forward implications (follow directly from (11)):

$$\begin{aligned}&\mathrm{OFF}_j \rightarrow \mathrm{ON}_i \quad \text{and} \quad \mathrm{OFF}_k \rightarrow \mathrm{ON}_i \\ &\mathrm{ON}_j \wedge \mathrm{ON}_k \rightarrow \mathrm{OFF}_i\end{aligned} \tag{12}$$

- Backward implications (follow from (11) and (5)):

$$\begin{aligned}&\mathrm{OFF}_i \rightarrow \mathrm{ON}_j \quad \text{and} \quad \mathrm{OFF}_i \rightarrow \mathrm{ON}_k \\ &\mathrm{ON}_i \wedge \mathrm{ON}_j \rightarrow \mathrm{OFF}_k \quad \text{and} \quad \mathrm{ON}_i \wedge \mathrm{ON}_k \rightarrow \mathrm{OFF}_j\end{aligned} \tag{13}$$

These implications are used to update the on- and off-sets of the node functions as follows:

- Forward updates:
  - If the off-set of input j expands (turning don't-cares to 0s), the on-set of i may expand (turning don't-cares to 1s). From (12):

$$\mathrm{ON}_i := \mathrm{ON}_i \vee \mathrm{OFF}_j \tag{14}$$

- Similarly, if the off-set of input k expands (turning don't-cares to 0s), the on-set of i may expand (turning don't-cares to 1s). From (12):

$$\mathrm{ON}_i := \mathrm{ON}_i \vee \mathrm{OFF}_k \tag{15}$$

- If the overlap of the on-sets of inputs j and k expands (turning don't-cares in either node to 1s), the off-set of i may expand (turning don't-cares to 0s). From (12):

$$\mathrm{OFF}_i := \mathrm{OFF}_i \vee \left(\mathrm{ON}_j \wedge \mathrm{ON}_k\right) \tag{16}$$

- Backward updates:
  - If the off-set of the output i expands, the on-set of both inputs may expand. From (13):

$$\mathrm{ON}_j := \mathrm{ON}_j \vee \mathrm{OFF}_i \text{ and } \mathrm{ON}_k := \mathrm{ON}_k \vee \mathrm{OFF}_i \tag{17}$$

  - If the overlap of the on-sets of output i and input j expands (turning don't-cares in either node to 1s), the off-set of the other input k may expand. From (13):

$$\mathrm{OFF}_k := \mathrm{OFF}_k \vee \left(\mathrm{ON}_i \wedge \mathrm{ON}_j\right) \tag{18}$$

  - Similarly, if the overlap of the on-sets of output i and input k expands (turning don't-cares in either node to 1s), the off-set of the other input j may expand. From (13):

$$\mathrm{OFF}_j := \mathrm{OFF}_j \vee \left(\mathrm{ON}_i \wedge \mathrm{ON}_k\right) \tag{19}$$

2.5 NAND2 Consistency Requirements

During the synthesis process we need to know what functions can be connected to the inputs of a NAND2 gate knowing the function at its output. Let's assume that we know the functions at output i and input k. We can solve for the permissible functions at input j by setting up the following "constraint" function:

$$\left(\mathrm{LB}_i \le f_i \le \mathrm{UB}_i\right) \wedge \left(\mathrm{LB}_k \le f_k \le \mathrm{UB}_k\right) \wedge \left(f_i = f_j' \vee f_k'\right) \tag{20}$$

This can be transformed to the normal form:

$$\left(\mathrm{LB}_i \wedge f_i'\right) \vee \left(f_i \wedge \mathrm{UB}_i'\right) \vee \left(\mathrm{LB}_k \wedge f_k'\right) \vee \left(f_k \wedge \mathrm{UB}_k'\right) \vee \left(f_i \oplus \left(f_j' \vee f_k'\right)\right) = 0 \tag{21}$$

Upon universally quantifying f_i and f_k, and applying Boole's expansion in terms of f_j, we obtain the following result:

$$\begin{aligned} \mathrm{LB}_j &= \mathrm{UB}_i' \\ \mathrm{UB}_j &= \left(\mathrm{LB}_k' \wedge \mathrm{UB}_i\right) \vee \left(\mathrm{LB}_i' \wedge \mathrm{UB}_k\right) \end{aligned} \tag{20}$$

This can be equivalently expressed as:

$$\begin{aligned} \mathrm{ON}_j &= \mathrm{OFF}_i \\ \mathrm{OFF}_j &= \mathrm{ON}_i \wedge \mathrm{ON}_k \end{aligned} \tag{21}$$

During the synthesis process, Equation (21) identifies the functions that can be connected at input j from the knowledge of the functions at the output and the other input. In particular, when the input k is unconnected its function $f_k \in [0,1]$. This leads to the following simplification of (21):

$$\begin{aligned} \mathrm{ON}_j &= \mathrm{OFF}_i \\ \mathrm{OFF}_j &= 0 \end{aligned} \tag{22}$$

For a node l in the network to be connected to the input j of our NAND2 gate, two conditions must be satisfied. First, the node l must be *structurally consistent* with our NAND2 gate node. This is the case if l does not appear in the transitive fan-out of i. Second, the node l must be *functionally consistent* with the functions at the output i and input k of this gate. To determine if the node l is *functionally consistent*, the function $f_l \in [\mathrm{LB}_l, \mathrm{UB}_l]$ of the node l must satisfy the following condition:

$$[\mathrm{LB}_l, \mathrm{UB}_l] \cap [\mathrm{LB}_j, \mathrm{UB}_j] \neq \varnothing \tag{23}$$

After some algebra, this requirement can be stated as follows:

$$\begin{aligned} &\mathrm{OFF}_l \wedge \mathrm{OFF}_i = 0 \\ &\text{and} \\ &\mathrm{ON}_l \wedge \mathrm{ON}_k \wedge \mathrm{ON}_i = 0 \end{aligned} \tag{24}$$

In other words,

- The intersection of the off-sets of f_l and f_i must be empty, and
- The intersection of the on-sets of f_l, f_k, and f_i must be empty.

The second requirement is vacuously satisfied if there is nothing connected at input k ($\mathrm{ON}_k = 0$).

The set of nodes in the network which are both structurally consistent and functionally consistent with a NAND2 gate node will make up the *connectible set* for this node.

2.6 Covering

At any given point during the synthesis process we have a partial network of NAND2 gates. The goal of the process is to find covers for the on-sets of all gates while observing three types of constraints:

1. IO Constraints, i.e., the fact that the functions at the primary input nodes and primary output gate nodes are given and cannot be changed by the synthesis process (of course if some of the output functions are not completely specified, their on- and off-sets will be extended by the synthesis process.)
2. Structural constraints that insure the evolving network remains acyclic
3. Functional constraints (24) that insure consistency among the functions of NAND2 gate nodes

A minterm m in the on-set of some gate node (i, j, k) is *covered* if the following condition is satisfied:

$$m \leq \mathrm{OFF}_j \vee \mathrm{OFF}_k \tag{25}$$

i.e., m is contained in the off-set of either input. The set of *uncovered minterms* for node (i, j, k) is

$$\mathrm{UnCovered}_i := \mathrm{ON}_i \setminus (\mathrm{OFF}_j \vee \mathrm{OFF}_k) = \mathrm{ON}_i \wedge \mathrm{OFF}_j' \wedge \mathrm{OFF}_k' \tag{26}$$

When $\mathrm{UnCovered}_i$ is empty for each gate node (i, j, k) in a network, the network is called *complete*: $\bigvee_{i \in \text{ gate nodes}} (\mathrm{UnCovered}_i = \varnothing)$. In a *partial network* at least one UnCovered set remains non-empty.

A node l which is structurally and functionally consistent with node i can be used to cover the minterm $m \in \mathrm{UnCovered}_i$, if the following condition is satisfied:

$$(m \leq \mathrm{OFF}_l \vee \mathrm{DC}_l) \equiv \left(m \wedge (\mathrm{OFF}_l \vee \mathrm{DC}_l)' = 0\right) \tag{27}$$

This condition takes into account the possibility of moving m from the don't-care set to the off-set of f_l. By noting that $(\mathrm{OFF}_l \vee \mathrm{DC}_l)' = \mathrm{ON}_l$, this condition can be simplified to:

$$m \wedge \mathrm{ON}_l = 0 \tag{28}$$

Covering a minterm $m \in \mathrm{UnCovered}_i$ can be accomplished in one of the following ways:

1. If either $j = 0$ or $k = 0$, i.e., an input of node i is unconnected, then it may be possible to find another node l which is *structurally and functionally consistent* with node i, such that *it covers or can be made to cover* minterm m. The set of candidate nodes that meet these criteria must therefore satisfy functional consistency (24) and include:

a. Any primary input node l which already covers m, i.e., $m \leq \mathrm{OFF}_l$.

b. Any existing gate node l which either covers m ($m \leq \mathrm{OFF}_l$) or can be made to cover m from its don't-care set ($m \leq \mathrm{DC}_l$).

c. A new gate node l which is "designed" to cover m: $\mathrm{ON}_l = 0$, $\mathrm{OFF}_l = m$.

2. If either j or k are non-zero, i.e., at least one input to the gate is already connected to an existing gate node, then we are assured that the structural and functional constraints have already been satisfied for this node and what remains to be done is to modify the off-set of the input to include m. The input $l \in \{j,k\}$ that can be modified in this way must satisfy $m \leq \mathrm{DC}_l$.

Let i be the output of a NAND2 gate, and m be a minterm in i's on-set. When a new gate j is added to i's input with the goal of covering this particular minterm, its function should be specified as follows

$$\begin{aligned} \mathrm{ON}_j &:= \mathrm{OFF}_i \\ \mathrm{OFF}_j &:= m \end{aligned} \tag{29}$$

When a new gate k is added to the other input of a NAND2 gate whose output is i and whose other input is j, its function should be specified as follows:

$$\begin{aligned} \mathrm{ON}_k &:= \mathrm{OFF}_i \\ \mathrm{OFF}_k &:= m \vee \mathrm{ON}_j \end{aligned} \tag{30}$$

Figure 2.3 summarizes the various formulas described here that will be used by the synthesis algorithm, BESS, given in Chapter 3.

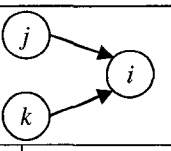
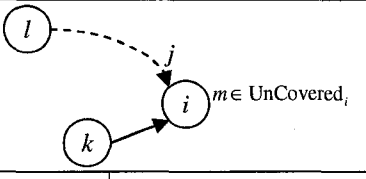
<table>
<tr><td colspan="4"> 
</td></tr>
<tr><td></td><td>NAND2 Constraint</td><td>$ON_i = OFF_j \vee OFF_k$</td><td>$OFF_i = ON_j \wedge ON_k$</td></tr>
<tr><td rowspan="2">Implications/Updates:</td><td>Forward</td><td>$ON_i = ON_i \vee OFF_j$
$ON_i = ON_i \vee OFF_k$</td><td>$OFF_i = OFF_i \vee (ON_j \wedge ON_k)$</td></tr>
<tr><td>Backward</td><td>$ON_j = ON_j \vee OFF_i$
$ON_k = ON_k \vee OFF_i$</td><td>$OFF_k = OFF_k \vee (ON_i \wedge ON_j)$
$OFF_j = OFF_j \vee (ON_i \wedge ON_k)$</td></tr>
<tr><td colspan="4"> 
</td></tr>
<tr><td rowspan="2">Consistency Requirements:</td><td>Structural</td><td colspan="2">$l \notin$ Transitive fanout of i</td></tr>
<tr><td>Functional</td><td>$ON_l \wedge ON_k \wedge ON_i = 0$</td><td>$OFF_l \wedge OFF_i = 0$</td></tr>
<tr><td rowspan="3">Coverage:</td><td>Constraint</td><td colspan="2">$m \wedge ON_l = 0$</td></tr>
<tr><td>Action</td><td>$OFF_l = OFF_l \wedge m$</td><td>$ON_l = ON_l \vee OFF_i$</td></tr>
<tr><td>Uncovered</td><td colspan="2">$UnCovered_i = ON_i \wedge OFF_l' \wedge OFF_k'$</td></tr>
</table>

Figure 2.3: Summary of Formulas used in the Synthesis Algorithm

Chapter 3

Optimal NAND2 Synthesis Algorithm

3.1 Algorithm Description

In this chapter we present the Branch-and-bound Exact Synthesis System (BESS). In this algorithm, the synthesis of a network will be completed incrementally, driven by the covering of minterms of the gate nodes in the network. These coverings are performed by (1) adding NAND2 gate nodes to the network, (2) adding connections between those nodes, and (3) updating the functions of the nodes. The synthesis of the network is completed when each on-set minterm of every gate node in the network is covered, resulting in a final network implementation of the specified function(s). The optimal network implementation of the function(s) is found by using a backtracking branch-and-bound search to explore all possible ways of performing these coverings. This branch-and-bound search method for performing optimal synthesis is based on similar work by Davidson [Davidson 68b] and Nakagawa [Nakagawa 89]. In this version, improvements to the techniques used for decision heuristics, functional implications, and pruning are made. We also provide an analysis of the conflicts that may occur within the search and provide a proof of convergence for the algorithm.

We begin the description of the NAND2 synthesis algorithm BESS by first presenting a simplified version. The motivation here is to illustrate the necessary concepts of the search. We will then discuss the issues that arise with this simplified version, which will lead to a more complete version of the algorithm.

3.1.1 Synthesis Example

The example provided in this section introduces the basic framework of BESS. The tables on the following 7 pages provide a detailed trace of the execution of the algorithm that will follow. In each table, the details of a single covering made in the algorithm are given for each decision.

The algorithm begins with a Boolean function (or set of Boolean functions) specified by the designer. In this example, a two-input XOR function is chosen for synthesis. Based on this function an initial partial network is produced. This network and the data associated with the nodes in the network are given in the Initialization step of the trace. The initial network contains a single output node (node 3) and two primary input nodes (nodes 1 and 2). The Boolean function associated with each input node is simply the primary input represented by the node. The on-, off-, and dc-sets of these functions are given in columns four

through six in the table. The Boolean function associated with the output node is the function chosen for synthesis ($x_1x_2 \vee x_1'x_2'$ in this example). The on-, off-, and dc-sets for this function are also provided in the table.

The synthesis algorithm is driven by the covering of minterms. Thus the uncovered set of minterms for each gate node must be maintained as the algorithm progresses. Since NAND2 gates are used as the building block for synthesis, the uncovered set is made up of minterms from the on-set of the gate node. In the initial network, the uncovered set for gate node 3 contains the entire set of on-set of minterms since no covering has yet been completed.

The first step of the algorithm is described by Decision 1. The minterm x_1x_2' from the uncovered set of gate node 3 is chosen for covering. Once a minterm is chosen for covering, a node must be selected to perform the covering. A node selected to complete the covering must be both functionally and structurally connectible to the selected gate node as described in Chapter 2. The first column in the table labeled "Struct" indicates the set of nodes in the network which are structurally connectible to this gate node. A node which is functionally connectible must satisfy two constraints: $\mathrm{OFF}_i \wedge \mathrm{OFF}_j$ and $\mathrm{ON}_i \wedge \mathrm{ON}_k \wedge \mathrm{ON}_i$. The results for these tests are given for each node in columns 9 and 10. Those nodes which are both functionally and structurally connectible will be indicated with a C in the "Conn" column of the table. Any node marked as connectible may be used to complete the covering.

In this first step, neither node 1 nor node 2 are functionally connectible to node 3. Therefore no existing node can be used to cover the minterm x_1x_2'. This implies the that a new gate node must be added to the network to perform the covering. We call this a structural implication.

The details of the covering using a new gate node are given at the bottom of the Decision 1 table. First a new gate node, node 4, is added to the network. This node initially has an unspecified Boolean function. In order to cover the selected minterm, x_1x_2', this minterm is added to the off-set of node 4 and this node must be connected to node 3 as a fan-in. Finally functional implications must be propagated through the network according to the rules outlined in Chapter 2. Here, the on-set and uncovered sets of node 4 are changed, in addition to the uncovered set of node 3.

Since both gate nodes 3 and 4 have uncovered minterms remaining, the network is not complete. Therefore the algorithm must continue. This leads to the second step of the algorithm described by Decision 2.

Once again an uncovered minterm is selected for covering. In this step, the minterm $x_1'x_2'$ from gate node 4 is chosen. Nodes 1 and 2 are found to be structurally connectible to the gate node 4, while only node 1 is functionally connectible. This implies that node 1 is the only existing node which can be used to cover the selected minterm. A new gate node can also be added to the network to complete the covering.

In this step, node 1 is chosen to complete the covering. Thus, node 1 is added as a fan-in of gate 4 and functional implications are propagated through the network

In the resulting network, both gate nodes 3 and 4 have uncovered minterms remaining. Therefore another covering must be completed. The details of this covering are given under Decision 3. This same process of covering is repeated until all on-set minterms from every gate node have been covered. In this example, 8 coverings are required. The network and final table given under Decision 8 provide the final NAND2 implementation for the 2-input xor function.

Figure 3.1 provided at the end of the trace shows the search tree traversed by the synthesis algorithm for this example. The nodes of the search tree represent the (node, minterm) pairs covered at a given decision step. The edges of the tree describe the connection made in the network to complete the covering. In some cases, multiple edges may leave a search tree node. This indicates that a decision was made about which node would be used to cover the minterm. In other cases, only a single edge leaves a search tree node. Here, no decision was necessary; an implication rather than a decision was made about the covering.

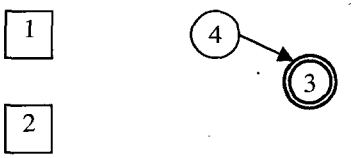
INITIALIZATION

Struct	Conn	Node l	ON_l	OFF_l	DC_l	$UnCovered_l$	Functionally Consistent? (0 means yes)		Can cover m?
							$OFF_l \wedge OFF_i$	$ON_l \wedge ON_k \wedge ON_i$	$m \wedge ON_l$
		1	x_1	x_1'	0	–			
		2	x_2	x_2'	0	–			
		3	$x_1'x_2 \vee x_1x_2'$	$x_1x_2 \vee x_1'x_2'$	0	$x_1'x_2 \vee x_1x_2'$			

Decision 1 Cover (3, $x_1'x_2$): $i = 3, m = x_1x_2'$

Struct	Conn	Node l	ON_l	OFF_l	DC_l	$UnCovered_l$	Functionally Consistent? (0 means yes)		Can cover m?
							$OFF_l \wedge OFF_i$	$ON_l \wedge ON_k \wedge ON_i$	$m \wedge ON_l$
S		1	x_1	x_1'	0	–	$\neq 0$	NA	NA
S		2	x_2	x_2'	0	–	$\neq 0$	NA	NA
	*	3	$x_1'x_2 \vee x_1x_2'$	$x_1x_2 \vee x_1'x_2'$	0	$x_1'x_2 \vee x_1x_2'$			

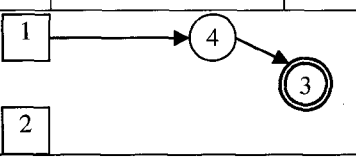
- Add new gate: $ON_4 = OFF_4 = 0$ (Structural Implication)
- Cover m: $OFF_4 := OFF_4 \vee m = x_1x_2'$
- Connect gate and propagate updates (functional implications)
- Backward: $ON_4 := ON_4 \vee OFF_3 = x_1x_2 \vee x_1'x_2'$
- $UnCovered_4 := ON_4 = x_1x_2 \vee x_1'x_2'$
- $UnCovered_3 := ON_3 \wedge OFF_4' \wedge (0)' = x_1'x_2$



Decision 2 Cover (4, $x_1'x_2'$): $i = 4, m = x_1'x_2'$

Struct	Conn	Node l	ON_l	OFF_l	DC_l	$UnCovered_l$	Functionally Consistent? (0 means yes)		Can cover m?
							$OFF_l \wedge OFF_i$	$ON_l \wedge ON_k \wedge ON_i$	$m \wedge ON_l$
S	C	1	x_1	x_1'	0	–	0	0	0
S		2	x_2	x_2'	0	–	$\neq 0$	NA	NA
		3	$x_1'x_2 \vee x_1x_2'$	$x_1x_2 \vee x_1'x_2'$	0	$x_1'x_2$			
	*	4	$x_1x_2 \vee x_1'x_2'$	x_1x_2'	$\neq 0$	$x_1x_2 \vee x_1'x_2'$			

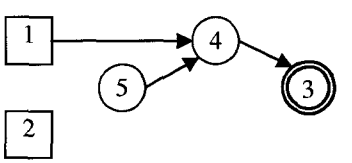
- Two choices: node 1 and a new gate - Connect 1 and propagate updates
- Forward: $ON_4 := ON_4 \vee OFF_1 = x_1' \vee x_2$
- $UnCovered_4 := ON_4 \wedge OFF_1' \wedge (0)' = x_1x_2$



Decision 3 Cover (4, x_1x_2): $i = 4, m = x_1x_2$

Struct	Conn	Node l	ON_l	OFF_l	DC_l	$UnCovered_l$	Functionally Consistent? (0 means yes)		Can cover m?
							$OFF_l \wedge OFF_i$	$ON_l \wedge ON_k \wedge ON_i$	$m \wedge ON_l$
S		1	x_1	x_1'	0	–	0	$\neq 0$	NA
S		2	x_2	x_2'	0	–	$\neq 0$	NA	NA
		3	$x_1'x_2 \vee x_1x_2'$	$x_1x_2 \vee x_1'x_2'$	0	$x_1'x_2$			
	*	4	$x_1' \vee x_2$	x_1x_2'	0	x_1x_2			

- Add new gate: $ON_5 = OFF_5 = 0$ (Structural Implication)
- Cover m: $OFF_5 := OFF_5 \vee m = x_1x_2$
- Connect gate and propagate updates (functional implications)
- Backward: $ON_5 := ON_5 \vee OFF_4 = x_1x_2'$
- $UnCovered_5 := ON_5 = x_1x_2'$
- $UnCovered_4 := ON_4 \wedge OFF_5' \wedge OFF_1' = 0$

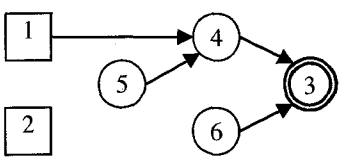


Decision 4 Cover (3, $x_1'x_2$): $i = 3, m = x_1'x_2$

Struct	Conn	Node l	ON_l	OFF_l	DC_l	$UnCovered_l$	Functionally Consistent? (0 means yes)		Can cover m?
							$OFF_l \wedge OFF_i$	$ON_l \wedge ON_k \wedge ON_i$	$m \wedge ON_l$
S		1	x_1	x_1'	0	–	$\neq 0$	NA	NA
S		2	x_2	x_2'	0	–	$\neq 0$	NA	NA
	*	3	$x_1'x_2 \vee x_1x_2'$	$x_1x_2 \vee x_1'x_2'$	0	$x_1'x_2$			
S		4	$x_1' \vee x_2$	x_1x_2'	0	0	0	$\neq 0$	NA
S		5	x_1x_2'	x_1x_2	$\neq 0$	x_1x_2'	$\neq 0$	NA	NA

- Add new gate: $ON_6 = OFF_6 = 0$ (Structural Implication)
- Cover m: $OFF_6 := OFF_6 \vee m = x_1'x_2$
- Connect gate and propagate updates (functional implications)
- Backward: $ON_6 := ON_6 \vee OFF_3 = x_1'x_2' \vee x_1x_2$

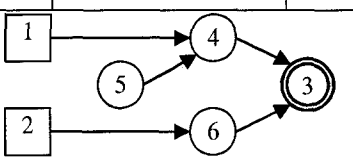
  $OFF_6 := OFF_6 \vee (ON_4 \vee ON_3) = x_1'x_2$
- $UnCovered_6 := ON_6 = x_1'x_2' \vee x_1x_2$
- $UnCovered_3 := ON_3 \wedge OFF_4' \wedge OFF_6' = 0$



Decision 5 Cover (6, $x_1'x_2'$): $i = 6, m = x_1'x_2'$

Struct	Conn	Node l	ON_l	OFF_l	DC_l	$UnCovered_l$	Functionally Consistent? (0 means yes)		Can cover m?
							$OFF_l \wedge OFF_i$	$ON_l \wedge ON_k \wedge ON_i$	$m \wedge ON_l$
S		1	x_1	x_1'	0	–	$\neq 0$	NA	NA
S	C	2	x_2	x_2'	0	–	0	0	0
		3	$x_1'x_2 \vee x_1x_2'$	$x_1x_2 \vee x_1'x_2'$	0	0			
S		4	$x_1' \vee x_2$	x_1x_2'	0	0	0	0	$\neq 0$
S	C	5	x_1x_2'	x_1x_2	$\neq 0$	x_1x_2'	0	0	0
	*	6	$x_1x_2 \vee x_1'x_2'$	$x_1'x_2$	$\neq 0$	$x_1x_2 \vee x_1'x_2'$			

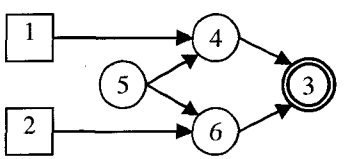
- Three choices: nodes 2, 5, and a new gate - Connect 2 and propagate updates
- Forward: $ON_6 := ON_6 \vee OFF_2 = x_1 \vee x_2'$
- $UnCovered_6 := ON_6 \wedge OFF_2' \wedge (0)' = x_1x_2$



Decision 6 Cover (6, x_1x_2): $i = 6, m = x_1x_2$

Struct	Conn	Node l	ON_l	OFF_l	DC_l	$UnCovered_l$	Functionally Consistent? (0 means yes)		Can cover m?
							$OFF_l \wedge OFF_i$	$ON_l \wedge ON_k \wedge ON_i$	$m \wedge ON_l$
S		1	x_1	x_1'	0	–	$\neq 0$	NA	NA
S		2	x_2	x_2'	0	–	0	$\neq 0$	NA
		3	$x_1'x_2 \vee x_1x_2'$	$x_1x_2 \vee x_1'x_2'$	0	0			
S		4	$x_1' \vee x_2$	x_1x_2'	0	0	0	$\neq 0$	NA
S	C	5	x_1x_2'	x_1x_2	$\neq 0$	x_1x_2'	0	0	0
	*	6	$x_1 \vee x_2'$	$x_1'x_2$	0	x_1x_2			

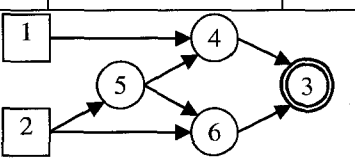
- Two choices: node 5, and a new gate - Connect 5 and propagate updates
- Forward: $ON_6 := ON_6 \vee OFF_5 = x_1 \vee x_2'$
- Backward: $OFF_5 := OFF_5 \vee (ON_2 \wedge ON_6) = x_1x_2$
  $ON_5 := ON_5 \vee OFF_6 = x_1x_2' \vee x_1'x_2$
- $UnCovered_6 := ON_6 \wedge OFF_2' \wedge OFF_5 = 0$
- $UnCovered_5 := ON_5 = x_1x_2' \vee x_1'x_2$



Decision 7 Cover (5, x_1x_2'): $i = 5, m = x_1x_2'$

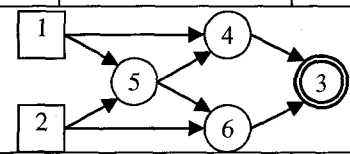
Struct	Conn	Node l	ON_l	OFF_l	DC_l	$UnCovered_l$	Functionally Consistent? (0 means yes)		Can cover m?
							$OFF_l \wedge OFF_i$	$ON_l \wedge ON_k \wedge ON_i$	$m \wedge ON_l$
S		1	x_1	x_1'	0	–	0	$\neq 0$	NA
S	C	2	x_2	x_2'	0	–	0	0	0
		3	$x_1'x_2 \vee x_1x_2'$	$x_1x_2 \vee x_1'x_2'$	0	0			
		4	$x_1' \vee x_2$	x_1x_2'	0	0			
	*	5	$x_1x_2' \vee x_1'x_2$	x_1x_2	$\neq 0$	$x_1x_2' \vee x_1'x_2$			
		6	$x_1 \vee x_2'$	$x_1'x_2$	0	0			

- Two choices: node 2 and a new gate - Connect 2 and propagate updates
- Forward: $ON_5 := ON_5 \vee OFF_2 = x_1' \vee x_2'$
- $UnCovered_5 := ON_5 \wedge OFF_2' \wedge (0)' = x_1'x_2$



Decision 8 Cover (5, $x_1'x_2$): $i = 5, m = x_1'x_2$

Struct	Conn	Node l	ON_l	OFF_l	DC_l	$UnCovered_l$	Functionally Consistent? (0 means yes)		Can cover m?
							$OFF_l \wedge OFF_i$	$ON_l \wedge ON_k \wedge ON_i$	$m \wedge ON_l$
S	C	1	x_1	x_1'	0	–	0	0	0
S		2	x_2	x_2'	0	–	0	$\neq 0$	NA
		3	$x_1'x_2 \vee x_1x_2'$	$x_1x_2 \vee x_1'x_2'$	0	0			
		4	$x_1' \vee x_2$	x_1x_2'	0	0			
		5	$x_1' \vee x_2'$	x_1x_2	$\neq 0$	$x_1'x_2$			
	*	6	$x_1 \vee x_2'$	$x_1'x_2$	0	0			
• Two choices: node 1 and a new gate - Connect 1 and propagate updates • Forward: $ON_5 := ON_5 \vee OFF_1 = x_1' \vee x_2'$ • $UnCovered_5 := ON_5 \wedge OFF_2' \wedge OFF_1' = 0$									
		1	x_1	x_1'	0	–			
		2	x_2	x_2'	0	–			
		3	$x_1'x_2 \vee x_1x_2'$	$x_1x_2 \vee x_1'x_2'$	0	0			
		4	$x_1' \vee x_2$	x_1x_2'	0	0			
		5	$x_1' \vee x_2'$	x_1x_2	$\neq 0$	0			
		6	$x_1 \vee x_2'$	$x_1'x_2$	0	0			



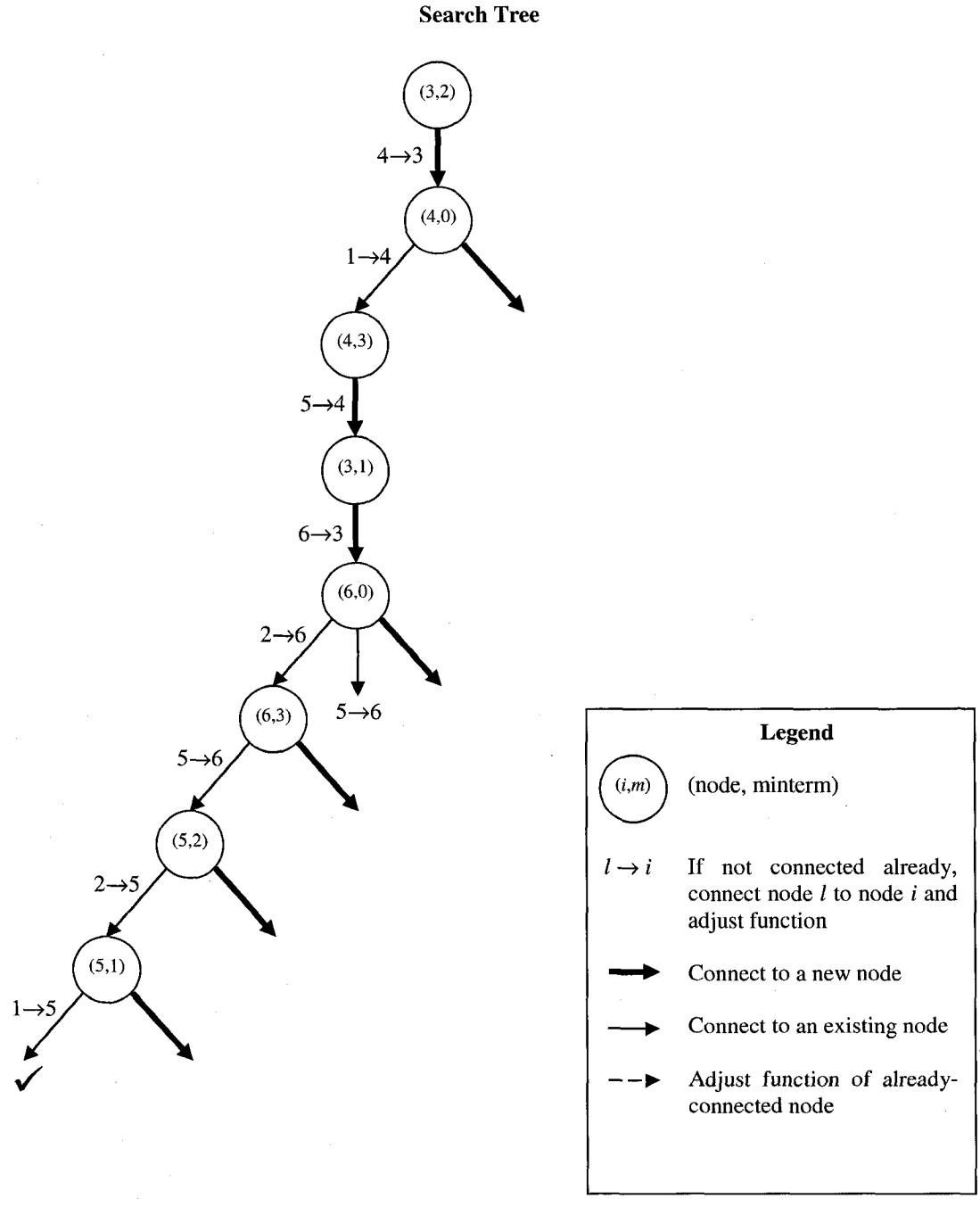


Figure 3.1: Trace and Search Tree for the Synthesis of a 2-input XOR function using NAND2 gates

3.1.2 Data Structures

Two principal data structures are required by BESS. The network structure is used to keep track of all the nodes contained in the network while a node substructure is used to maintain the more detailed information for the individual nodes in the network. Most of the interconnection information is maintained using the node substructure. The network structure only needs to store the nodes as three sets: primary output gate nodes, primary input nodes, and internal gate nodes. The network data structure will also store the value of the cost function for the network it describes. Details for the network data structure are given in Figure 3.2.

The node substructure maintains all the data needed to describe a particular node. This includes the Boolean function associated with the node (stored in terms of the on- and off-sets), and the type of node (whether the node is an input node or gate node). The node structure also maintains the data describing the relationships between this node and others in the network. The two inputs to the node are stored as the Left and Right inputs. The set of fan-out nodes and the set of connectible nodes are also stored by the node structure. Finally, the node structure will keep a Boolean function that describes the uncovered on-set minterms at this node. This uncovered function will determine whether synthesis on this node has been completed.

Network Structure	
Set of Nodes:	OutputNodes
Set of Nodes:	InputNodes
Set of Nodes:	InternalNodes
Integer:	Cost
Node Structure	
Boolean Function:	ON
Boolean Function:	OFF
Node Type:	Input or Gate
Node Input:	Left
Node Input:	Right
Set of Nodes:	Fan-out
Set of Nodes:	Connectible
Boolean Function:	UnCovered

Figure 3.2: Data Structures: Network structure and Node substructure

3.1.3 Initial Network

The input to the synthesis problem is a set of p completely- or partially-specified n-variable Boolean functions. When synthesis begins, the network will contain only n primary input nodes and p primary output gate nodes. The remainder of the nodes in the network will be added as synthesis progresses.

As an example, the initial network for the set of functions $\{f_1(x_1,x_2)=x_1 \oplus x_2,\ f_2(x_1,x_2)=x_1 \wedge x_2\}$ is shown in Figure 3.3. This network contains one primary input node for every variable used in the functions (2 in this case) and one primary output gate node for each function (also 2 in this case).

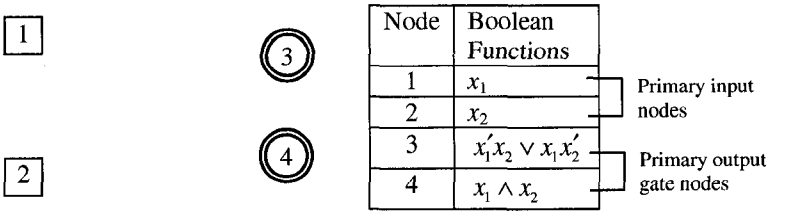


Figure 3.3: Initial Network for the set of functions $\{f_1(x_1,x_2) = x_1 \oplus x_2,\ f_2(x_1,x_2) = x_1 \wedge x_2\}$

3.1.4 Synthesis Procedure

The core procedure of BESS performs a single covering step, namely it covers one arbitrarily chosen uncovered minterm in the partial network. This main computational step performs a covering of some uncovered minterm in the partial network. Once the covering is made, a recursive call to the same procedure on the new network will continue the incremental synthesis of the network. Pseudocode for the SynthesizeNetwork procedure and two of its subprocedures is given in Figure 3.4.

The first step of the synthesis procedure is to determine if the network has been completed. If the on-set of every gate node in the network is covered, the network is complete. Therefore it can be stored as the solution network and the procedure can terminate.

If there are nodes remaining in the network that are not fully covered by their inputs then the network is not yet complete. The procedure will attempt to cover an uncovered minterm from at least one node. The covering process begins by first selecting an uncovered minterm, *m*, and its corresponding node, (*i, j, k*), from the network. The minterm that is chosen by this procedure is guaranteed to be covered during this synthesis step. It may be the case that additional uncovered minterms in the network are covered as well. This will depend on how the initial covering is completed

Once an uncovered minterm is selected, a node, *l*, is chosen which can be used to cover this minterm. This node must satisfy the consistency and covering requirements described in Section 2.6. It can be an existing node in the network or a new gate node added to the network specifically to perform this covering. The node *l* should be chosen from *i*'s connectible set to ensure that it is both functionally and structurally consistent as an input of *i*. In addition it must also satisfy the covering constraint: $m \wedge \mathrm{ON}_l = 0$.

Depending on the type of node *l*, the PerformCovering procedure will make different changes to the network.

1. If *l* is a **primary input node**, then it is not yet connected to *i*. Therefore the edge (*l*, *i*) must be added to the network. Since *l* is a primary input node, *l*'s Boolean function is completely specified. Therefore the only uncovered portion of *i* that *l* can cover is the set of minterms: $\mathrm{OFF}_l \wedge \mathrm{UnCovered}_i$. The function $\mathrm{OFF}_l \wedge \mathrm{UnCovered}_i$ must contain *m* since *l* was selected by the procedure to perform the covering. Therefore no change is required to the off-set of *l*.

2. If l is a **gate node** that already exists as an **input of** i, then the edge (l, i) already exists in the network. Thus the only change that must be made is to add m to l's off-set.

3. If l is an **existing gate node** that is not an input of i, then an edge (l, i) must be added to the network. Since l was chosen to cover the minterm m, $m \not\leq \mathrm{ON}_l$. However, it is possible that $m \leq \mathrm{DC}_l$. If this is the case, then m must be added to l's off-set.

4. If l is a **new gate node** that is not yet connected within the network, then the edge (l, i) must be added to the network. Since the global function of a new gate is initially the interval [0,1], l will not yet cover any minterm in i. Therefore the minterm m must be added to the off-set of l.

With these changes complete, the network returned by the PerformCovering procedure will be a network that has the minterm $m \leq \mathrm{ON}_i$ covered by the node l.

The process of covering m may add an edge between the nodes i and l and may change the function of node l. Due to these changes, functional implications must be made throughout the network to keep the functions at the inputs and output of the gate nodes consistent. Both forward and backward updates are made according to the rules given in Section 2.4:

Forward updates: $\mathrm{ON}_i := \mathrm{ON}_i \vee \mathrm{OFF}_j$, $\mathrm{ON}_i := \mathrm{ON}_i \vee \mathrm{OFF}_k$, and $\mathrm{OFF}_i := \mathrm{OFF}_i \vee (\mathrm{ON}_j \wedge \mathrm{ON}_k)$

Backward updates: $\mathrm{ON}_j := \mathrm{ON}_j \vee \mathrm{OFF}_i$, $\mathrm{ON}_k := \mathrm{ON}_k \vee \mathrm{OFF}_i$, $\mathrm{OFF}_k := \mathrm{OFF}_k \vee (\mathrm{ON}_i \wedge \mathrm{ON}_j)$, and

$\mathrm{OFF}_j := \mathrm{OFF}_j \vee (\mathrm{ON}_i \wedge \mathrm{ON}_k)$

Using these implications, the changes made by a covering can be propagated throughout the network so that once complete, every gate node will satisfy the on- and off-set constraints. This propagation is done by traversing through the network beginning at node (i, j, k) and making the necessary updates to the Boolean function at i and then proceeding to the inputs and outputs of i.

Due to the functional changes that may occur in the network, a change in the uncovered set of the node i may also result. The uncovered set for a node may increase if the on-set of the node i increased, while the uncovered set may decrease if some minterms are covered by either j or k. For this reason, $\mathrm{UnCovered}_i$ should be updated as the on- and off-sets of node i are updated.

Once the functional implications have been completed, the procedure updates the connectible set of each node in the network. Since the Boolean functions may have changed for some nodes, the set of nodes that are now connectible may have also changed. It is possible that previously covered nodes have now become uncovered, and therefore the set of connectible nodes will need to be computed for these nodes as well. The connectible set of each uncovered gate node in the network can be found according to the functional and structural consistency requirements described in Section 2.5. Now that the covering has been completed, the same covering procedure is repeated on the new partial network to cover another

uncovered minterm. The procedure terminates when the uncovered sets of all gate nodes have become empty.

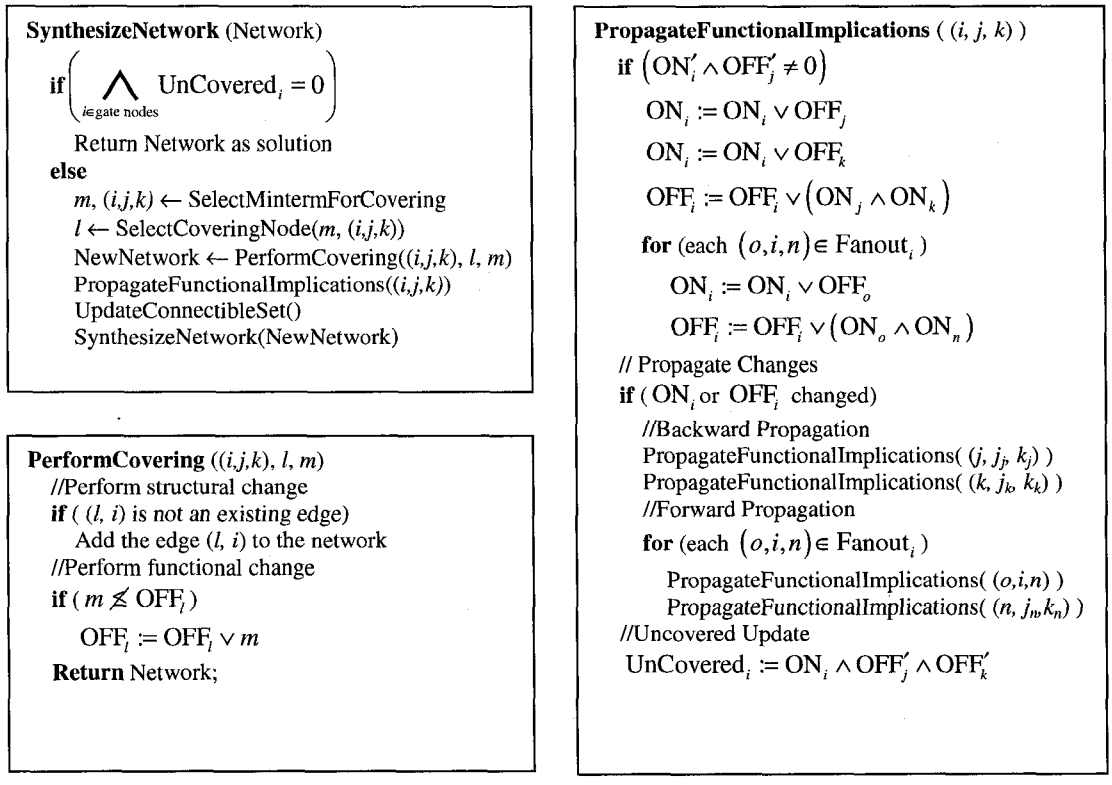


Figure 3.4: Pseudocode for NAND2 Synthesis Algorithm

3.1.5 Algorithm Discussion

There are several issues that arise with this initial version of the algorithm that must be addressed:

1. **Optimality:** The optimality of the network is not guaranteed by this version of the algorithm. However, the basic computational steps described in this initial version can be used in a branch-and-bound backtrack search method. By searching the entire space of complete networks which implement the desired function(s) the optimal cost implementation can be determined. Section 3.2 describes how this search is performed.

2. **Decision Strategies**: The synthesis procedure must make two decisions during its execution. The first is the choice of which uncovered on-set minterm will be selected for covering in this step. The second is the choice of which node will be used to cover this minterm. The choices made at these decision points will effect both the final network produced by the algorithm as well as the number of coverings that will be required to complete the network. In Section 3.3 we provide an investigation into how these choices effect the synthesis process and then use this analysis to provide a set of heuristics that can be used to make these decisions.

3. **Decision Implications**: In many search procedures, the assignment of a value to a decision variable produces a chain of implications in the resulting problem instance. Here, both structural and functional implications will result once a covering has been completed. One type of functional implication was already described in the initial version of the algorithm. Additional non-local functional implications are also possible based on certain structures that may exist in the network. In addition, structural implications may also be possible. These implications effect the basic structure of the network rather than the functions at the nodes.

4. **Conflicts**: Since two choices must be made for each covering (both the minterm to be covered and the way in which it is covered) it is possible that the set of choices made may lead to a conflict. In BESS a conflict occurs when an inconsistent Boolean function appears at the output or input of a NAND2 gate node. We will discuss in Section 3.5 how these inconsistencies appear and how the algorithm handles the situation when they do.

5. **Search Space**: In Section 3.6 we analyze the search space of the algorithm. This space is quite unique compared to other search algorithms in that it is constantly changing as synthesis proceeds. Thus it is difficult to obtain bounds on the run-time and size of the search. However, we will discuss two representations of the search space and then provide an analysis of its size based on these representations.

6. **Termination**: As with any search algorithm, a proof of convergence is necessary to guarantee that the search will terminate and an optimal solution will be found. The convergence proof for this branch-and-bound search algorithm is not as trivial as many such proofs are for similar types of search algorithms. This is due to the fact the search space is constantly changing. The proof we provide in Section 3.7 takes into account these changes while providing a guarantee that the search will terminate.

7. **Completeness**: Just as the proof of convergence is complicated by the changing search space, so is the proof completeness. Thus, we give a detailed proof in Section 3.8 that provides a guarantee that the algorithm will produce the optimal network with respect to the specified cost function (the number of gates in the network).

8. **Pruning**: Due to the way that decisions are made in the search and certain properties of the networks being created, we have found that some partial networks may be regenerated at different points in the search. A significant portion of the search space can be pruned by detecting and removing these repeated networks. Section 3.9 describes how partial networks can be repeated and what can be done to prevent them from occurring.

3.2 Branch-and-Bound Backtrack Search

By using the main computational steps described in the initial version of the algorithm (Section 3.1.4) a branch-and-bound backtrack search method can be created which will search for an optimal-cost network within the set of all network implementations.

In the SynthesizeNetwork procedure given in Figure 3.4 the choice of the covering node *l* directs the algorithm towards a specific network implementation for the set of primary output functions. If an alternate covering option was used instead to perform this same covering, an alternate network implementation would result. Thus, a backtracking algorithm which generates all possible network implementations for a set of primary output functions can be created simply by extending the current SynthesizeNetwork procedure to explore each option that exists for covering a selected minterm. The optimal cost network can then be determined by comparing the number of gates in each network. The pseudocode for such an extension is provided in Figure 3.5.

```
SynthesizeNetwork v.2 (Network)
  if ( ∧ (i∈gate nodes) UnCovered_i = 0 )
    if (number gates in Network is smallest seen so far)
      Store Network as current minimum
  else
    m, (i,j,k) ← SelectMintermForCovering
    CovNodes ← FindAllCoveringNodes(m, (i,j,k))
    for (all l ∈ CovNodes)
      NewNetwork ← PerformCovering((i,j,k), l, m)
      PropagateFunctionalImplications((i,j,k))
      UpdateConnectibleSet()
      SynthesizeNetwork(NewNetwork)
```

Figure 3.5: Pseudocode for Optimal NAND2 Synthesis Algorithm – Version 2

While this algorithm will lead to an optimal network, it is possible to perform some pruning of the search through a bounding technique. As synthesis of a network progresses, coverings are made in the network. As we described in Section 3.1.4 there are four ways that a covering can be completed. In each case the cost of the network, i.e. the number of gates in the network, either increases or remains the same. Thus, the cost of any complete network which can be created by completing a given partial network must have cost greater than or equal to this partial network. This idea leads to the following bounding technique which can be employed in this synthesis search.

Bounding Technique: Let S be the set of all complete networks which can be created by completely synthesizing a given partial network N. A lower bound on the cost of the networks in S is the cost of the partial network N.

Using this bounding technique, the SynthesizeNetwork procedure can be extended so that it will perform a branch-and-bound search. The updated pseudocode is provided in Figure 3.6.

```
SynthesizeNetwork v.3 (Network)
if ( ⋀_{i∈gate nodes} UnCovered_i = 0 )
   if (number gates in Network is smallest seen so far)
      Store Network as current minimum
      UpperBound ← Cost_Network
else
   m, (i,j,k) ← SelectMintermForCovering
   CovNodes ← FindAllCoveringNodes(m, (i,j,k))
   for (all l ∈ CovNodes)
      NewNetwork ← PerformCovering((i,j,k), l, m)
      PropagateFunctionalImplications((i,j,k))
      UpdateConnectibleSet()
      if (Cost_NewNetwork < UpperBound)
         SynthesizeNetwork(NewNetwork)
```

Figure 3.6: Pseudocode for Branch-and-Bound version of Optimal NAND2 Synthesis Algorithm – Version 3

3.3 Decision Strategies

The synthesis procedure makes two decisions during its execution. The first is the choice of which uncovered on-set minterm will be selected for covering in this step. The second is the choice of which node will be used to cover this minterm. In this section we investigate how these choices effect the order in which complete networks are generated by the algorithm and the number of covering steps required for the synthesis of a particular complete network. The insights provided through this investigation will then lead to a set of heuristics that can be used to aid in making these decisions.

3.3.1 Minterm Selection

By definition, every on-set minterm of every gate node in the network must be covered before the network is considered complete. However, the choice of which minterm to cover first can greatly effect the size of the search space. The choice of the minterm can effect both the number of coverings needed to complete a network, i.e. *the height of this path in the search tree*, and the number of choices available for performing a covering, i.e. the *width of this branch of the search tree*. For these reasons a heuristic should be employed which will attempt to select the minterm that will minimize both the height and width of the search tree under this branch. Therefore a correlation between the minterms available for covering and the resulting search trees based on these coverings must be discovered.

First the correlation between the selection of a minterm and the width of the branch will be considered. The branching factor for a node in the search tree is the number of options that exist for covering the selected minterm. In order to keep the width of the branches small, a minterm should be selected which has as few covering options as possible. Fewer choices for covering will lead to a smaller branch width and a higher probability that the optimal solution is represented by the selected partial network.

The selection of a minterm can also effect the height of the search tree. The height of the search tree can be reduced when minterms are covered by functional implications. When choosing a minterm for covering, the possibility that this minterm may be covered later as a result of an alternate minterm covering must be considered. Since covering a minterm in one node in the network can cause functional implications throughout the entire network, a minterm from a node which is least likely to become covered through functional implications should be chosen first.

Based on the relationships between the size of the search tree and the minterms available for covering several options exists for heuristic methods which will help to determine which uncovered minterm is the best choice for covering. Many of these heuristics can be used in combination to produce a ranking of the minterms.

SmallestCOV: From the set of uncovered minterms, select the minterm that has the *fewest covering options.*

DifficultCOV: From the set of uncovered minterms, select the minterm which has the most difficult *covering rank*: The four methods for performing a covering can be ranked according to their difficulty. From easiest to the most difficult:

1. Gate node that already exists as an input
2. Primary input node
3. Existing gate node
4. New gate node

Covering a minterm with a node that already exists in the fan-in set of a node is the easiest type of covering for two reasons: (1) It is likely that this minterm could have been covered as part of functional implications resulting from previous coverings. (2) Very little change is made in the network when this covering is performed putting little constraint on the network. The most difficult covering is performed by adding a new gate. By adding a new gate to the network the cost increases and a whole new set of minterms must be covered.

Using this list, minterms can be ranked according to their easiest covering option. The minterm with the most difficult covering label will then be chosen for covering first.

SmallestFI: From the list of uncovered nodes, select the node that has the *smallest fan-in.* (A node may have 0,1,or 2 nodes in its fan-in). From the set of uncovered minterms for this node, choose an arbitrary minterm to cover.

SmallestCONN: From the list of uncovered nodes, select the node that has the *smallest connectible set.* From the set of uncovered minterms for this node, choose an arbitrary minterm to cover.

Each of these heuristic methods uses an individual property of the nodes and minterms that remain to be covered in order to make a selection. However, these individual properties do not provide enough granularity to distinguish between all possible uncovered minterms. Therefore it is beneficial to use multiple heuristic methods in combination.

For example, the SmallestFI method can be used to select the set of uncovered nodes which have the fewest number of input nodes. This set can then be narrowed down using the SmallestCONN method to those nodes with the connectible sets of smallest size. Finally an uncovered minterm from this set of nodes can be chosen according to the SmallestCOV method.

The effectiveness of these heuristics is explored in Chapter 4.

3.3.2 Covering Node Decision

The choice of which node will be used to perform a covering drives the structure of the final complete network. Even though, the entire set of complete networks is searched by the branch-and-bound version of this synthesis algorithm, this decision is still important. It effects the order in which the complete networks are explored. In an ideal situation, the partial network representing an optimal network will always be chosen first for further synthesis. This way, the cost bound is set to the optimal value initially and later networks can be pruned earlier based on the bounding condition.

The optimal networks are not known in advance however. Therefore we cannot know which partial network will represent an optimal solution as the search is progressing. For this reason, we create heuristic methods that can be used to select a covering option that is *more likely* to put partial networks representing optimal solutions towards the beginning.

CovOrder: Order the nodes available for covering according to their *covering rank*: (1) a primary input node, (2) a gate node that already exists as a fan-in, (3) an existing gate node that is not connected as a fan-in, (4) a new gate node. Nodes with lower covering rank are chosen first.

CostOrder: Order the nodes available for covering according to the cost of the partial networks that result when the node is used for covering. Networks with lower cost are chosen first

The CostOrder heuristic is more expensive but can potentially lead to better decisions since each new partial networks must be created to determine the ordering. Once again, the effectiveness of these heuristic methods is discussed in Chapter 4.

3.4 Decision Implications

Once a decision is made about the covering that is to be performed, implications of this covering can be propagated through the networks. These implications provide additional information about functional and structural properties of the network, which help to make further decisions in the next set of covering steps.

3.4.1 Functional Implications

Functional implications are one type of implication that can result from a covering. In this case, minterms of the don't-care set are moved to the on- and off-sets of gate nodes. These implications depend on the structure of the network and the Boolean functions at the nodes in the network.

The set of local functional implications were described in the initial description of the synthesis algorithm (Section 3.1.4). These implications are used to maintain the consistency of the functions at the immediate output and inputs of a gate node.

3.4.2 Global Functional Implications

Two types of global functional implications will be considered here. Both are based on a network structure containing reconvergent fan-out. These global function implications are similar to the recursive learning algorithm described by Kunz [Kunz 94].

3.4.2.1 Global Functional Implication 1: Simple Structure

The first type of global functional implication will be based on the simple reconvergent structure shown in Figure 3.7. Here, the fan-out of node 4 travels along two paths through nodes 2 and 3 until it converges again at node 1. This structure is called *simple* since the two paths from 4 to 1 contain only 3 nodes.

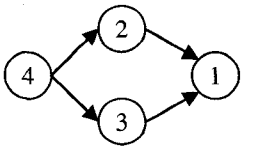


Figure 3.7: Simple Reconvergent Pattern

The functional implication $ON_1 \rightarrow ON_4$ that results from this structure is based on the fact that the fan-in set of node 1 has reached the limit of two nodes. Therefore no additional node can be added to the fan-in of 1 to cover the uncovered minterms from 1. Thus, there are only two ways that minterms from the on-set of 1 can be covered.

Let m be a minterm from the on-set of node1. If node 2 were to cover this minterm, then m must be contained in the on-set of 2. Then by the backward implication rule: $OFF_2 \rightarrow ON_4$ the minterm m would also be contained in the on-set of 4. Alternately, if node 3 was used to cover this minterm, then m must be contained in the off-set of 3. By the same backward implication rule: $OFF_3 \rightarrow ON_4$ the minterm m would

once again be contained in the on-set of 4. Thus every minterm in the on-set of 1 must appear in the on-set of 4, implying the implication rule: $ON_1 \rightarrow ON_4$.

This functional implication rule can be used whenever a simple reconvergent fan-out structure exists in a partial network. As with the local functional implications, this one implication may lead to further functional implications in the network.

3.4.2.2 Global Functional Implication 2: General Structure

The second type of global functional implication is based on a more general reconvergent fan-out structure which extends through more nodes as shown in Figure 3.8. Once again, the fan-out of node 4 travels along two distinct paths until it converges again at node 1. In this general structure, however, the paths can be any length.

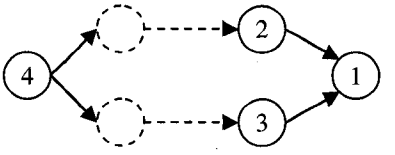


Figure 3.8: General Reconvergent Pattern

A simple functional implication rule is not possible in this case due to the variability in the structure of the network. It is possible however that changes can be made to the Boolean function of some node in this structure based on the reconvergent pattern. This type of learning is completed by temporarily performing alternate coverings at node 1 and examining the consequences of these coverings.

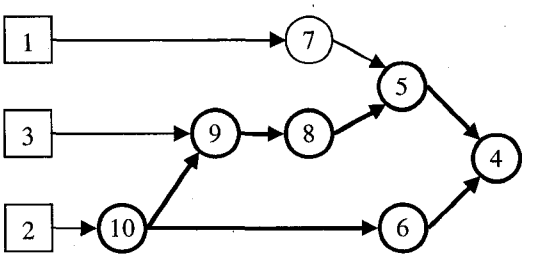
This type of functional implication is best described with an example. The network given in Figure 3.9 contains the general reconvergent fan-out structure from Figure 3.8. The fan-out of node 10 converges again at node 4. The two paths that form this reconvergence are $(10,9,8,5,4)$ and $(10,6,4)$.

The fan-in set of node 4 has reached the maximum of two inputs which implies that there are only two ways that the three remaining uncovered minterms of 4 can be covered.

If node 5 is used to cover these minterms then each minterm would be added to off-set of 5 and then functional implications would be propagated through the network resulting in the following functional changes: $ON_8 = x_2' x_3 \vee x_1$ and $ON_7 = x_1' \vee x_2 \vee x_3'$. If node 6 is used to cover these minterms then each minterm would be added to the off-set of 6 and these changes would be propagated through the networks using the functional implication rules. The result this time would be: $ON_{10} = x_1 \vee x_2' \vee x_3'$, $OFF_9 = x_2' x_3 \vee x_1 x_3$, and $ON_8 = x_2' x_3 \vee x_1 x_3$

If the functions that result from the two possible coverings are compared, the minterm $x_1 x_2 x_3$ appears in the on-set of node 8 in both cases while it does not appear in the on-set of 8 in the original network.

Therefore a functional implication exists where the on-set of 8 is expanded to include $x_1x_2x_3$: $ON_8 = x_2'x_3 \vee x_1x_2$.



Node l	ON_l	OFF_l	$UnCovered_l$
1	x_1	x_1'	-
2	x_2	x_2'	-
3	x_3	x_3'	-
4	$x_1'x_2'x_3 \vee x_2x_3' \vee x_1x_3' \vee x_1x_2$	$x_1'x_2'x_3' \vee x_1'x_2x_3 \vee x_1x_2'x_3$	$x_1x_2 \vee x_1x_3'$
5	$x_1'x_3' \vee x_1'x_2 \vee x_1x_2'x_3$	$x_1'x_2'x_3$	0
6	$x_1'x_2'x_3' \vee x_1'x_2x_3 \vee x_1x_2'x_3$	$x_1'x_2x_3'$	$x_1'x_2'x_3' \vee x_1x_2'x_3$
7	x_1'	$x_1x_2'x_3$	0
8	$x_2'x_3$	$x_1'x_3' \vee x_1'x_2$	0
9	$x_3' \vee x_1'x_2$	$x_2'x_3$	0
10	$x_2' \vee x_1'x_3'$	$x_1'x_2x_3$	$x_1'x_2x_3'$

Choice 1: 5 covers minterms from 4

$M = x_1x_2x_3 \vee x_1x_2x_3' \vee x_1x_2'x_3'$

$OFF_5 := OFF_5 \vee M = x_1'x_2'x_3 \vee x_1x_2 \vee x_1x_3'$

Propagate Local Implications:

Backward: $ON_8 := ON_8 \vee OFF_5 = x_2'x_3 \vee x_1$

Backward:

$ON_7 := ON_7 \vee OFF_5 = x_1' \vee x_2 \vee x_3'$

Choice 2: 6 covers minterms from 4

$M = x_1x_2x_3 \vee x_1x_2x_3' \vee x_1x_2'x_3'$

$OFF_6 := OFF_6 \vee M = x_2x_3' \vee x_1x_2 \vee x_1x_3'$

Propagate Local Implications:

Backward: $ON_{10} := ON_{10} \vee OFF_6 = x_1 \vee x_2' \vee x_3'$

Forward: $OFF_9 := OFF_9 \vee (ON_3 \vee ON_{10}) = x_2'x_3 \vee x_1x_3$

Forward: $ON_8 := ON_8 \vee OFF_9 = x_2'x_3 \vee x_1x_3$

Global Implications

$OFF_5 = (OFF_5 \text{ from Choice 1}) \wedge (OFF_5 \text{ from Choice 2}) = (x_1'x_2'x_3 \vee x_1x_2 \vee x_1x_3') \wedge (x_1'x_2'x_3) = x_1'x_2'x_3$	no change
$OFF_6 = (OFF_6 \text{ from Choice 1}) \wedge (OFF_6 \text{ from Choice 2}) = (x_1x_2x_3') \wedge (x_2x_3' \vee x_1x_2 \vee x_1x_3') = x_1x_2x_3'$	no change
$ON_7 = (ON_7 \text{ from Choice 1}) \wedge (ON_7 \text{ from Choice 2}) = (x_1' \vee x_2 \vee x_3') \wedge (x_1') = x_1'$	no change
$ON_8 = (ON_8 \text{ from Choice 1}) \wedge (ON_8 \text{ from Choice 2}) = (x_2'x_3 \vee x_1) \wedge (x_2'x_3 \vee x_1x_2) = x_2'x_3 \vee x_1x_2$	**Func. Impl.**
$OFF_9 = (OFF_9 \text{ from Choice 1}) \wedge (OFF_9 \text{ from Choice 2}) = (x_2'x_3) \wedge (x_2'x_3 \vee x_1x_3) = x_2'x_3$	no change
$ON_{10} = (ON_{10} \text{ from Choice 1}) \wedge (ON_{10} \text{ from Choice 2}) = (x_2' \vee x_1'x_3') \wedge (x_1 \vee x_2' \vee x_3') = x_2' \vee x_1'x_3'$	no change

Figure 3.9: Finding Functional Implications based on General Reconvergent Pattern

Implications based on this general type of reconvergent fan-out are more difficult to find than those from the simpler structure. However, the reasoning behind why these implications can be made remains the same. To find a general reconvergent structure all nodes whose fan-in set contains the maximum of two nodes must be considered. The set of nodes which are descendents for each fan-in are then found. If one node is common to both of these descendent sets, a reconvergent structure exists. Once the structure is found, implications can be discovered by testing both covering options and comparing the functions of the nodes within the structure as in the example above.

3.4.2.3 Global Functional Implication Procedures

A new version of the **SynthesizeNetwork** procedure which includes both types of global implications is given in Figure 3.10.

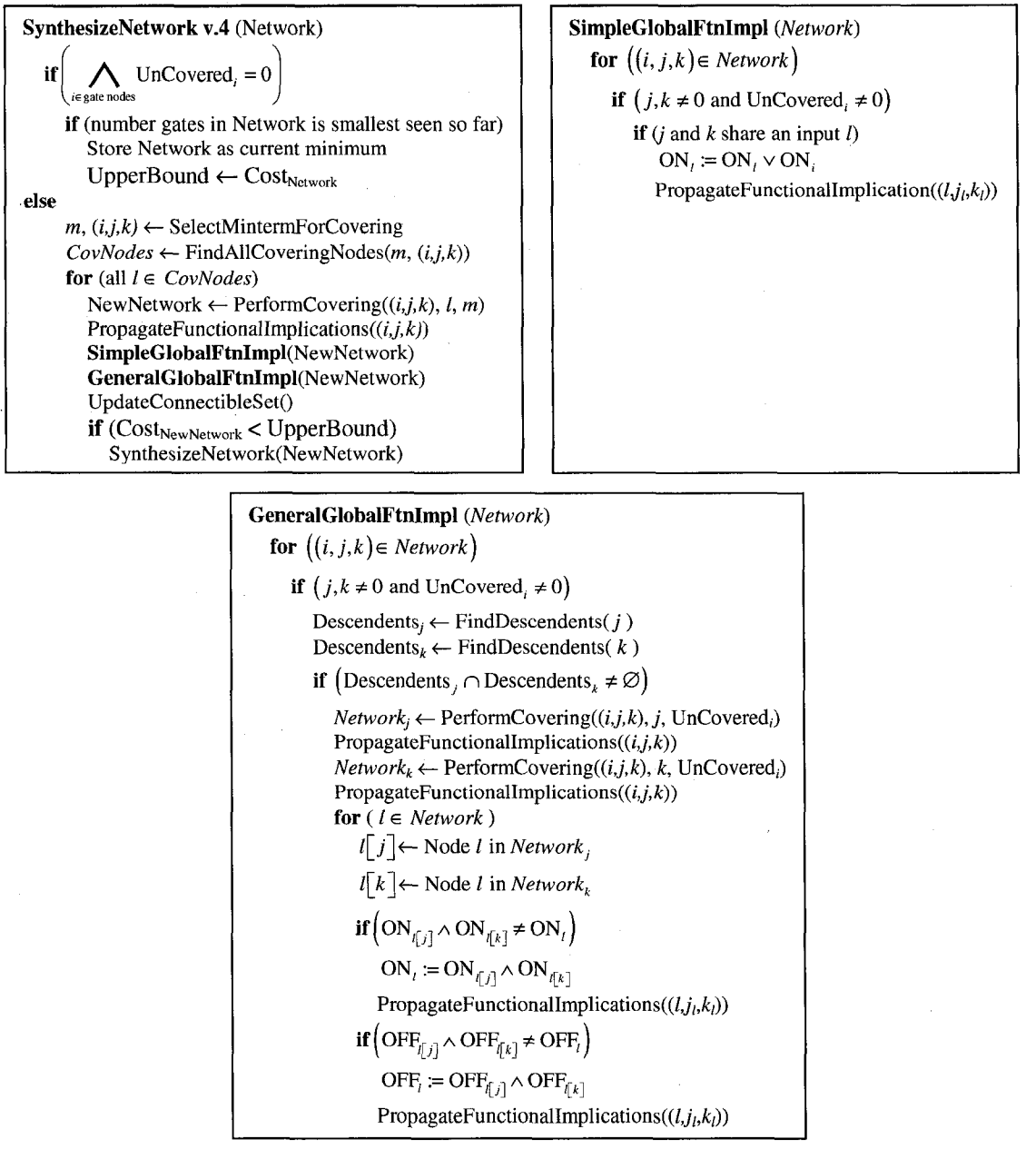


Figure 3.10: Synthesize Network Function – Version 4 and Global Functional Implication Procedures

3.4.3 Structural Implications

Structural implications can also result from a covering decision. Additional details about the partial network are learned based on the current state of the network.

As we saw in the example given in Section 3.1.1, it is often the case that no existing node can cover a minterm (Decisions 1, 3, and 4). The only "choice" for covering in this situation is to add a new gate to the network to cover the minterm. This choice can be viewed as a *structural implication* of the previous covering since no choice remains as to how the minterm can be covered.

The additional information learned by completing structural implications immediately will often allow partial networks to be pruned earlier based on their cost. This, in turn, can reduce the search required to find an optimal network. The example given in Section 3.1.1 can be reduced from 8 decisions to 5.

The SynthesizeNetwork procedure must be changed to include structural implications. After the procedure propagates the functional implications of the covering, it then updates the connectible sets of the gate nodes. Now a procedure AddNewGate can be called which will determine if any structural implications are possible and will add a new gate to the network for one of the structural implications that can be performed. As long as new gates are added to the network, there remains potential for additional structural implications. Therefore the two steps of computing the connectible sets and performing structural implications should be repeated until the network remains unchanged.

The AddNewGate procedure detects and performs a single structural implication in a partial network. First detection of an implication must be completed. Then the covering is completed as in the PerformCovering function described earlier. The new gate node is added to the network and is used to cover a minterm from the selected node that cannot be covered by any existing node.

Notice that structural implications must be completed individually. It is often the case that when a gate node is added to the network it can be used to cover many of the minterms that previously could not be covered by an existing node. Thus, if a new gate node were added for every uncovered minterm an optimal network may be missed.

More than one structural implication may be required. If the added node is not connectible to an existing node which contains minterms that cannot be covered then a second call to the AddNewGate procedure may be required. In addition, if an on-set minterm in the added node is not coverable by an existing node then a structural implication will exist at the new gate node and an additional call to the AddNewGate procedure will be needed to cover the minterm.

Structural implications can also be performed as part of an initialization step for the algorithm. When the initial partial network is created, it is possible that one or more of the output functions cannot be completely covered by only input nodes. In this case, at least one new gate node must be added to the network to cover these nodes. No complete networks will be eliminated by these structural implications since every network will require these additional gates. This initialization can be performed by calling the AddNewGate procedure on the original partial network before the main SynthesizeNetwork procedure is begun.

```
SynthesizeNetwork v.5 (Network)
  if ( ∧_{i∈ gate nodes} UnCovered_i = 0 )
    if (number gates in Network is smallest seen so far)
      Store Network as current minimum
      UpperBound ← Cost_Network
  else
    m, (i,j,k) ← SelectMintermForCovering
    CovNodes ← FindAllCoveringNodes(m, (i,j,k))
    for (all l ∈ CovNodes)
      NewNetwork ← PerformCovering((i,j,k), l, m)
      PropagateFunctionalImplications((i,j,k))
      SimpleGlobalFtnImpl(NewNetwork)
      GeneralGlobalFtnImpl(NewNetwork)
      do
        UpdateConnectibleSet()
        AddNewGate (NewNetwork)
      while ( nodes were added to NewNetwork)
      if (Cost_NewNetwork < UpperBound)
        SynthesizeNetwork(NewNetwork)
```

```
AddNewGate (Network)
  //Detect uncoverable node
  for ((i, j, k) ∈ Network)
    Uncoverable ← UnCovered_i
    for (l ∈ {j, k, Connectible_i})
      Uncoverable = Uncoverable ∧ ON_l
    if (Uncoverable ≠ 0)
      Store i as the uncovered node;

  // Perform Covering
  Create a new gate node (l,0,0) in Network;
  Add the edge (l,i) to the network
  OFF_l := SelectMinterm(Uncoverable)
  PropagateFunctionalImplications((i,j,k))
```

Figure 3.11: Synthesize Network Procedure – Version 5 and Structural Implication Procedures

3.5 Conflicts

Conflicts can arise during the search because certain combinations of covering choices may result in a network which contains one or more gate nodes with an inconsistent Boolean function. The propagation of functional implications after a covering is performed can help to eliminate many covering choices that would lead to conflicts. Despite these efforts however, conflicts can still occur.

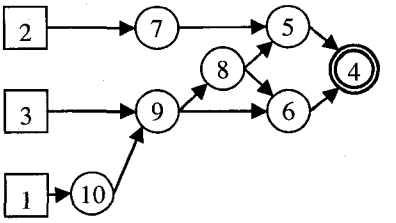
Figure 3.12 provides a situation in which a conflict occurs. The first partial network contains a node 7 which is connectible to 8 according to the connectibility constraints and will cover the minterm $x_1'x_2x_3'$ in 8. To perform this covering, the edge (7,8) is added to the network and the resulting functional implications are propagated through the network. The partial network that results however is invalid. The function at node 4 is inconsistent since the intersection of the on- and off-sets is non-empty.

When an invalid network is detected in the search, the algorithm will simply stop searching this portion of the search tree and backtrack to the previous decision. In the case of the example from Figure 3.12, the algorithm would try adding a new gate to the network to cover node 8, since this second alternative had not yet been considered.

Notice that the function of a node in a fan-out free network can never become inconsistent as long as the local functional implication rules and consistency requirements are used. It is the reconvergent fan-out of the more complex networks which allow consistent coverings to be made only to have an inconsistency revealed through functional implications.

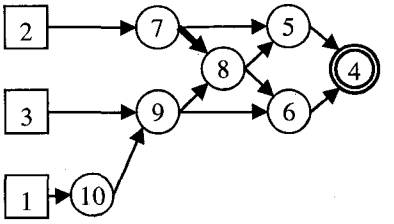
Decision Cover $(8, x_1'x_2x_3')$: $i = 8$, $m = x_1'x_2x_3'$

Struct	Conn	Node	ON_l	OFF_l	$UnCovered_l$
S		1	x_1	x_1'	–
S		2	x_2	x_2'	–
S		3	x_3	x_3'	–
		4	$x_1'x_2'x_3 \vee x_1'x_2x_3' \vee x_1x_2'x_3' \vee x_1x_2x_3$	$x_1'x_2'x_3' \vee x_1'x_2x_3 \vee x_1x_2'x_3 \vee x_1x_2x_3'$	$x_1x_2'x_3' \vee x_1x_2x_3$
		5	$x_1'x_3' \vee x_1'x_2 \vee x_2x_3' \vee x_1x_2'x_3$	$x_1'x_2'x_3$	0
		6	$x_1'x_2' \vee x_1'x_3 \vee x_2'x_3 \vee x_1x_2x_3'$	$x_1'x_2x_3'$	0
S	C	7	x_2'	$x_1'x_2$	0
	*	8	$x_1'x_3 \vee x_1'x_2$	$x_1x_2x_3' \vee x_1'x_2'x_3' \vee x_1x_2'x_3$	$x_1'x_2x_2'$
S		9	$x_3' \vee x_1x_2'$	$x_1'x_3$	0
S		10	x_1'	$x_1x_2'x_3$	0



- Two choices: node 7 and a new gate - Connect 7
  - Backward: $ON_7 := ON_7 \vee OFF_8 = x_2' \vee x_1x_3'$
  - Forward: $OFF_8 := OFF_8 \vee (ON_7 \wedge ON_9) = x_1x_3' \vee x_1x_2' \vee x_2'x_3'$
  - Forward: $ON_5 := ON_5 \vee OFF_8 = x_3' \vee x_1'x_2 \vee x_1x_2'$
  - Forward: $ON_6 := ON_6 \vee OFF_8 = x_2' \vee x_1'x_3 \vee x_1x_3'$
  - Forward: $OFF_4 := OFF_4 \vee (ON_5 \wedge ON_6) = x_2'x_3' \vee x_1'x_2x_3 \vee x_1x_2' \vee x_1x_3'$

		1	x_1	x_1'	–
		2	x_2	x_2'	–
		3	x_3	x_3'	–
		4	$x_1'x_2'x_3 \vee x_1'x_2x_3' \vee x_1x_2'x_3' \vee x_1x_2x_3$	$x_2'x_3' \vee x_1'x_2x_3 \vee x_1x_2' \vee x_1x_3'$	$x_1x_2'x_3' \vee x_1x_2x_3$
		5	$x_3' \vee x_1'x_2 \vee x_1x_2'$	$x_1'x_2'x_3$	0
		6	$x_2' \vee x_1'x_3 \vee x_1x_3'$	$x_1'x_2x_3'$	0
		7	$x_2' \vee x_1x_3'$	$x_1'x_2$	0
		8	$x_1'x_3 \vee x_1'x_2$	$x_1x_2'x_3' \vee x_1x_2x_3$	$x_1'x_2x_2'$
		9	$x_3' \vee x_1x_2'$	$x_1'x_3$	0
		10	x_1'	$x_1x_2'x_3$	0



Conflict: Inconsistent Function
$ON_4 \cap OFF_4 = x_1x_2'x_3'$

Figure 3.12: Example of Conflict: an Invalid Network

3.6 Search Space

There are two ways to view the search space of this algorithm. First it can be seen as the space of decisions that must be made to complete a network. There are a set of (node, minterm) pairs that require

covering (the variables in the decisions) and the alternate ways of performing these coverings (the values to be assigned). A solution is found when all minterms have been covered. The difference between this algorithm and that of a typical decision tree algorithm is that the set of variables is constantly changing. This makes it difficult to analyze the size of the search space based on the initial problem description since the number of minterms that must be covered can increase as networks are constructed.

The algorithm can also be seen as a way to systematically generate complete network implementations of the chosen function(s). As the algorithm uses the covering process to build a network, it is also searching through a space of circuits. The covering decision is used to divide the space of complete networks. An exhaustive search of this space must be completed in order to guarantee the optimality of the solution.

While the first view of the search space follows naturally from the description of the algorithm, the second is useful for evaluating the properties of the search. In this section, we will elaborate on the details of the search following both views, so that these concepts can be used later in discussions of the algorithm. Finally, we will use the first view to evaluate the size of the search space.

3.6.1 Search Space Version 1

BESS performs a search for an optimal network using a backtracking method. As with many backtrack search algorithms, the solution space is explored by assigning all possible values to the decision variables. In this search, the decision variables are the (node, minterm) pairs that require covering in the partial network. The values that can be assigned to these variables are the alternate ways that the covering of the (node, minterm) pair can be completed.

At the beginning of the search, the set of decision variables will contain only those uncovered minterms which come from primary output nodes. As the search progresses, (node, minterm) pairs are selected for covering. Then a covering is performed within the network and the (node, minterm) is removed from the set of decision variables. When a new gate is added to the network to complete this covering, or when functional implications are propagated through the network, new (node, minterm) pairs are then added to the set of decision variables. The algorithm continues synthesis on this single network until the network is complete – the set of decision variables is empty. From here, the algorithm will backtrack to a previous decision and create a new network based on an alternate covering for the selected node and minterm. This pattern of single network synthesis and backtracking will continue until all covering options for every (node,minterm) decision have been exhausted. This guarantees that every possible network has been explored.

If the cost of the smallest network seen so far is stored as subsequent networks are being constructed, the algorithm can use the cost to prune the search space at a partial network. The synthesis of a network can be stopped if the cost of this partial network exceeds the cost of the smallest complete network seen so far. This is possible since the cost of any complete network generated by the current partial network will

be greater than or equal to the cost of the partial network. This pruning allows us to reduce the size of the search space while still guaranteeing that an optimal network will be found.

This backtrack search is unique since the set of decision variables will both shrink and grow during the search. While the same (node, minterm) pair will never be added to the set of decision variables once it has been removed, it will be the case that additional (node, minterm) pairs will be added to the set of initial decision variables. This property causes difficulty when analyzing the size of the search space based on the initial problem description since the number of decision variables is not known at the beginning of the search.

3.6.2 Search Space Version 2

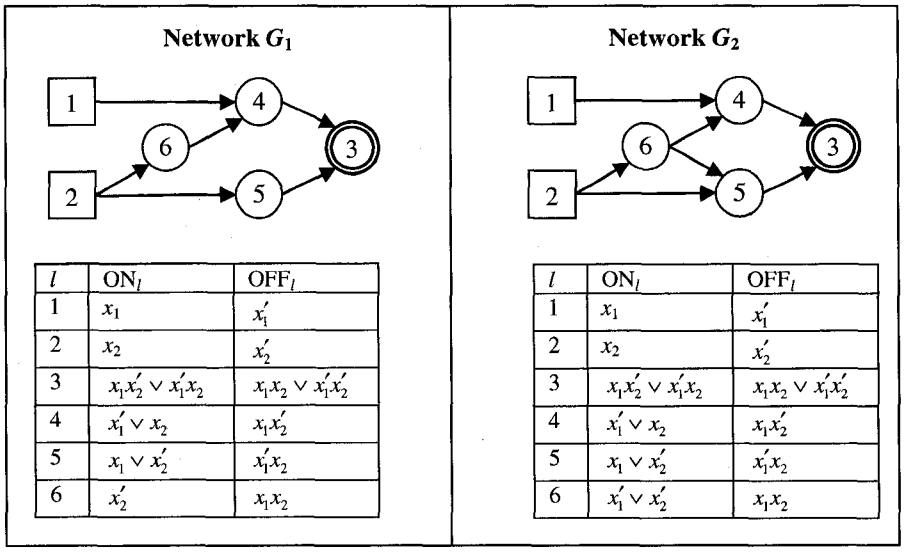
This description of the search space must begin with the basic definitions for various types of networks produced by the algorithm along with relationships that can exist between these networks.

A network is *valid* if it satisfies the functional and structural constraints:

1. The network must be acyclic.
2. The functions at the inputs and outputs of a node must be consistent.
3. The functions at the nodes representing the outputs of the network must be equal to the desired output functions.

Valid networks are categorized into two types. A *complete network* is a valid network in which every node is completely covered by its inputs, while a *partial network* is a valid network that has at least one node which remains uncovered. The functions of all gate nodes in a complete network are required to be completely specified.

Valid networks can be related to each other based on similarities in their node sets, edge sets, and global functions. A valid network $G_1=(V_1, E_1)$ is a *subnetwork* of the valid network $G_2=(V_2, E_2)$ if $V_1 \subseteq V_2$, $E_1 \subseteq E_2$, and for every node $i \in V_1$ there exists a corresponding node $j \in V_2$ such that $\mathrm{ON}_i \leq \mathrm{ON}_j$ and $\mathrm{OFF}_i \leq \mathrm{OFF}_j$. In this same case, G_2 is a *supernetwork* of G_1. This subnetwork relationship implies that G_1 can be expanded to become G_2. The network G_1 in Figure 3.13 is a subnetwork of G_2.



l	ON_l	OFF_l
1	x_1	x_1'
2	x_2	x_2'
3	$x_1x_2' \vee x_1'x_2$	$x_1x_2 \vee x_1'x_2'$
4	$x_1' \vee x_2$	x_1x_2'
5	$x_1 \vee x_2'$	$x_1'x_2$
6	x_2'	x_1x_2

l	ON_l	OFF_l
1	x_1	x_1'
2	x_2	x_2'
3	$x_1x_2' \vee x_1'x_2$	$x_1x_2 \vee x_1'x_2'$
4	$x_1' \vee x_2$	x_1x_2'
5	$x_1 \vee x_2'$	$x_1'x_2$
6	$x_1' \vee x_2'$	x_1x_2

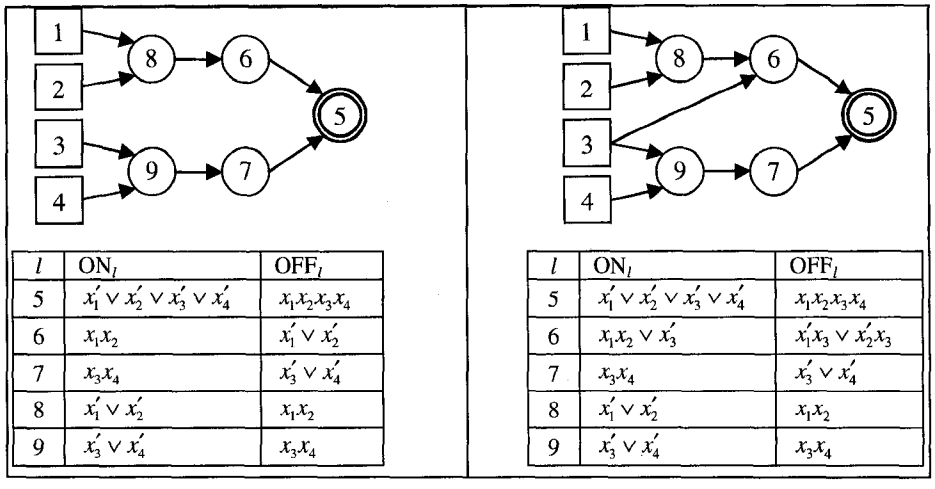
Figure 3.13: G_1 is a Functional Subnetwork of G_2

If G_1 is a subnetwork of G_2 and $G_1 \neq G_2$ then G_1 is a *proper subnetwork* of G_2. This implies that there must be either some node or edge that exists in G_2 that does not exist in G_1, or the global function of a node in G_1 is not as defined as the corresponding node in G_2.

Based on the proper subnetwork relationship, the category of complete valid networks is divided further into elementary and non-elementary networks. A complete valid network will be considered *elementary* if every proper structural subnetwork is not complete (i.e. the network is non-redundant). Similarly, a complete valid network is *non-elementary* if there exists a proper subnetwork which is also complete (i.e. the network is redundant). An *optimal* network will be an elementary network which has minimum cost.

By dividing the set of complete networks in this way, a limit can be placed on the algorithm so that it will search only elementary networks for an optimal network. At least one network with minimum cost must be elementary since any non-elementary network with minimum cost will always have a complete proper subnetwork with cost less than or equal to it.

The two networks given in Figure 3.14 are complete networks. The network in (a) is elementary. Every proper structural subnetwork is not complete. The network in (b) is non-elementary. Notice that the network in (a) is a proper structural subnetwork of this network and it is complete.



l	ON_l	OFF_l
5	$x_1' \vee x_2' \vee x_3' \vee x_4'$	$x_1x_2x_3x_4$
6	x_1x_2	$x_1' \vee x_2'$
7	x_3x_4	$x_3' \vee x_4'$
8	$x_1' \vee x_2'$	x_1x_2
9	$x_3' \vee x_4'$	x_3x_4

l	ON_l	OFF_l
5	$x_1' \vee x_2' \vee x_3' \vee x_4'$	$x_1x_2x_3x_4$
6	$x_1x_2 \vee x_3'$	$x_1'x_3 \vee x_2'x_3$
7	x_3x_4	$x_3' \vee x_4'$
8	$x_1' \vee x_2'$	x_1x_2
9	$x_3' \vee x_4'$	x_3x_4

(a) Elementary Network **(b) Non-elementary Network**

Figure 3.14: Elementary Networks

The search space for the exact synthesis algorithm is the set of all elementary networks that implement the desired functions. The algorithm will search this space for an optimal network using a branch-and-bound procedure that maintains a search tree. A node in the search tree corresponds to a partial network. A partial network is simply a representation of all complete networks of which it is a subnetwork. Therefore the search tree node corresponding to some partial network P represents the set of all elementary networks from the search space of which P is a subnetwork.

The root of the search tree corresponds to an initial partial network. This initial network must be a subnetwork for every elementary network in the search space. A leaf in the search tree corresponds to an elementary network. The set represented by an elementary network will be the singleton set containing only this elementary network.

The branching part of the branch-and-bound algorithm occurs when a covering is performed in the partial network at a search tree node. The alternate ways of performing this covering will create the branches of the search tree. Each partial network that results from the covering will represent a child of this search tree node, and the set of elementary networks represented by the original partial network will be divided into subsets each represented by one of the new partial networks.

The algorithm explores a single path in the search tree until a leaf is reached. Once the leaf is reached, the algorithm backtracks to the previous decision point (a branch in the search tree) and then explores the next path in the search tree. This pattern of branch exploration and backtracking will continue until all paths have been exhaustively searched. Since the algorithm exhaustively searches the entire search space, it will generate every elementary complete network. Therefore, it can determine which elementary network has the smallest cost.

If the cost of the smallest network seen so far is stored as the search tree is constructed, the algorithm can use the cost to prune the search at a partial network. The algorithm will be able to stop exploring the subtree under a search tree node if the cost of the partial network representing this node is greater than the cost of the smallest elementary network seen so far. This is possible since the cost of every elementary network represented by the partial network will be greater than or equal to the cost of the partial network. Using this cost bound technique, the size of the search tree can be reduced, while still guaranteeing that the minimum cost network will be generated.

3.6.3 Search Space Analysis

In order to analyze the run-time of the algorithm a bound must be given on the size of the search tree produced by the algorithm for a given function. In general, the total number of internal nodes in a search tree is defined by $\frac{w^{h+1}-1}{w-1}$, where w is the width of the branches in the search tree and h is the height of the tree [Cormen 01].

3.6.3.1 Search Tree Width

In order to give a worst-case analysis of the search space, the maximum width of a branch in the search tree must be considered. The size of a branch in the search tree is determined by the number of covering options for the selected minterm. There are at most $n + g$ ways a single minterm could be covered, where n is the number of input nodes and g is the number of gate nodes in the network. However g is dependent on the number of nodes in the network at any given time. Therefore the width of any branch in the search tree will be at most $w = n + G_{max}$ where G_{max} is the maximum number of gates that exists in any partial network encountered during the entire search. Using this value for the width, the size of the search tree is bounded above by $\frac{(n+G_{max})^{h+1}-1}{n+G_{max}-1}$.

An alternate upper bound on the branch-width of the search tree can be obtained by considering the number of n-input Boolean functions that can cover a given minterm. Given an arbitrary minterm, only half of the 2^{2^n} possible functions that exist on n or fewer inputs (2^{2^n-1} functions) can cover the minterm. Using this value for a bound on the width of the search tree gives an upper bound on the size of the search tree: $\frac{\left(2^{2^n-1}\right)^{h+1}-1}{2^{2^n-1}-1}$. The advantage to this bound is that it is not dependent on any property of the network generated by the algorithm and can be determined without knowing the size of the optimal network.

3.6.3.2 Search Tree Height

In the worst case, the algorithm will require one branch to be made for every on-set minterm in the network. An upper bound on this number will be the number of gate nodes in the network G multiplied by

the maximum $2^n - 1$ minterms that may need covering per gate node. Thus the height of the search tree will be less than $G\left(2^n - 1\right)$. Since the gates cannot repeat Boolean functions, the maximum number of gates cannot exceed the number of possible Boolean functions using *n* or fewer variables. Thus, $G \leq 2^{2^n}$. Using this height value, a final bound on the size of the search tree can now be obtained:

$$\frac{\left(n+G_{max}\right)^{G_{max}\left(2^n-1\right)+1}-1}{n+G_{max}-1} = O\left((n+G_{max})^{2^n G_{max}}\right) \text{ or } \frac{\left(2^{2^n-1}\right)^{2^{2^n}(2^n-1)+1}-1}{2^{2^n-1}-1} = O\left(2^{2^{2^n}}\right).$$

3.6.3.3 Further Search Space Analysis

While this worst-case analysis gives an upper bound on the size of the search tree, the experimental results presented in Chapter 4 will show that this is a very loose over approximation of the actual size of the search tree. In Chapter 4, further discussion of the search space of the algorithm will be provided using experimental evidence.

3.7 Convergence

As with any search algorithm, a proof of convergence is necessary to guarantee that the search will terminate and an optimal solution will be found. An algorithm converges if it eventually halts after a finite number of steps. A guarantee of the convergence of BESS can be given by showing that the search tree produced by the algorithm is finite and the procedures performed at each step of the recursion halt.

Each of the sub-procedures in the SynthesizeNetwork procedure will complete after a finite number of steps. In particular, the recursive PropagateFunctionalImplications procedure will halt after a finite number of calls. Recursive calls will only be made by this procedure when minterms are moved from the don't-care set of a node to the on- or off-set. In a network with a finite number of nodes, there is only a finite number of minterms that can be moved. Therefore the procedures performed at each step of the recursion will halt after a finite number of steps.

All that is left to show then is that the search tree produced by the algorithm is finite. A finite search tree implies that both the width of each branch in the tree and the height of each path can be bounded above. The width of a branch is bounded by the number of options that exist for covering the selected minterm. An upper bound on this number is the number of nodes in the network.

If the bound on the cost of the network is finite then each path in the search tree must be of finite length. At least one uncovered minterm is covered with every call to the SynthesizeNetwork procedure. Since a minterm can never be uncovered once it has been covered, there is a finite upper bound on the number of steps that are needed to cover all uncovered minterms in a given network assuming no additional gate

nodes are added to the network. If a new gate node is added to a partial network then additional uncovered minterms will be added as well. However, since the bound on the cost of the network is finite, only a finite number of nodes can be added to the network. Thus a finite upper bound on the number of steps needed to cover these new uncovered minterms exists as well.

If the cost bound is not initially set to be some finite value, then the first path in the search tree will have an infinite cost bound. In this case new gates can continually be added to cover minterms in the network and this path in the search tree will never be completed. To prevent these situations, some simple rules can be added to the algorithm. These additions will guarantee that a first network will eventually be completed. Thus the initial path of the search tree will be finite and a finite value will be assigned to the cost bound.

There are three rules that must be followed by BESS to ensure that the initial path of the search tree will eventually lead to a complete network:

1. The initial complete network must be a fan-out free network (the fan-out of every gate node is 1).
2. The first node chosen to cover a selected minterm should be an existing node rather than a new gate node when possible.
3. When choosing an uncovered minterm for covering, preference should be given to a minterm which cannot be covered by a fan-in node.

The first rule ensures that no network along the initial path will become invalid. As we discussed in Section 3.5 fan-out free networks cannot become invalid due to the functional implications made after each covering step. Therefore forcing the initial network to be fan-out free will guarantee that no backtracks will be necessary prior to finding the first complete network.

In order to force the first complete network to be fan-out free, the algorithm must choose input nodes, fan-in nodes, and new gate nodes over non-fan-in gate nodes for the initial choice of covering. This will ensure that the fan-out of every gate node in the network will be one.

The second and third rules prevent the algorithm from producing an infinite chain of gate nodes in the first network. The two situations that produce these infinite chains of gates are best explained through examples. On the following two pages a trace of the algorithm is provided where an infinite chain of gates results.

In this trace, the 3-input OR function will be synthesized. The trace begins with an initial partial network containing 3 primary input nodes and a single output node. At each decision in the trace, a new gate is chosen to cover the selected minterm. In Decision 1, a new gate node, node 5, is used to cover the minterm $x_1x_2x_3$ from node 4. In Decision 2, a new gate node, node 6, is used to cover the minterm $x_1'x_2'x_3'$ from node 5. It is possible that this same type of decision could continue to be made. If the cost bound for the network is infinite (as it is initially) then the network will continue to grow without bound.

The second rule prevents this type of infinite chain from occurring at the beginning of the search by requiring that an existing gate node be used to cover a minterm first (where possible). The chain of gates shown in this trace would be stopped at Decision 2 where one of the input nodes would be chosen for covering.

Algorithm Trace: Infinite Chain of Gates

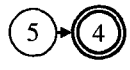
INITIALIZATION

Struct	Conn	Node	ON_l	OFF_l	$UnCovered_l$
		1	x_1	x_1'	–
		2	x_2	x_2'	–
		3	x_3	x_3'	–
		4	$x_1x_2x_3$	$x_1' \vee x_2' \vee x_3'$	$x_1x_2x_3$

Decision 1 Cover (4, $x_1x_2x_3$): $i = 4$, $m = x_1x_2x_3$

Struct	Conn	Node	ON_l	OFF_l	$UnCovered_l$
S		1	x_1	x_1'	–
S		2	x_2	x_2'	–
S		3	x_3	x_3'	–
	*	4	$x_1x_2x_3$	$x_1' \vee x_2' \vee x_3'$	$x_1x_2x_3$

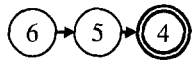
- Add new gate: $ON_5 = OFF_5 = \varnothing$ (Structural Implication)
- Cover *m*, connect gate and propagate updates



Decision 2 Cover (5, $x_1'x_2'x_3'$): $i = 5$, $m = x_1'x_2'x_3'$

Struct	Conn	Node l	ON_l	OFF_l	$UnCovered_l$
S	C	1	x_1	x_1'	–
S	C	2	x_2	x_2'	–
S	C	3	x_3	x_3'	–
		4	$x_1x_2x_3$	$x_1' \vee x_2' \vee x_3'$	0
	*	5	$x_1' \vee x_2' \vee x_3'$	$x_1x_2x_3$	$x_1' \vee x_2' \vee x_3'$

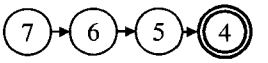
- Four choices: nodes 1, 2, 3, and a new gate
- Add a new gate: $ON_6 = OFF_6 = \varnothing$
- Cover *m*, connect gate, and propagate updates



Decision 3 Cover $(6,\ x_1x_2x_3)$: $i = 6$, $m = x_1x_2x_3$

Struct	Conn	Node l	ON_l	OFF_l	$UnCovered_l$
S		1	x_1	x_1'	–
S		2	x_2	x_2'	–
S		3	x_3	x_3'	–
		4	$x_1x_2x_3$	$x_1' \vee x_2' \vee x_3'$	0
		5	$x_1' \vee x_2' \vee x_3'$	$x_1x_2x_3$	$x_1'x_3 \vee x_2x_3' \vee x_1x_2'$
	*	6	$x_1x_2x_3$	$x_1'x_2'x_3'$	$x_1x_2x_3$

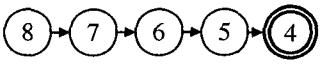
- Add new gate: $ON_7 = OFF_7 = \varnothing$ (Structural Implication)
- Cover m, connect gate, and propagate updates



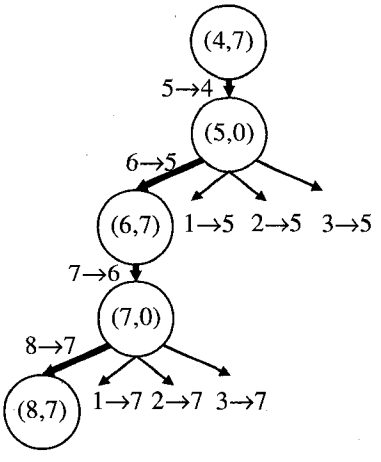
Decision 4 Cover $(7,\ x_1'x_2'x_3')$: $i = 7$, $m = x_1'x_2'x_3'$

Struct	Conn	Node	ON_l	OFF_l	$UnCovered_l$
S	C	1	x_1	x_1'	–
S	C	2	x_2	x_2'	–
S	C	3	x_3	x_3'	–
		4	$x_1x_2x_3$	$x_1' \vee x_2' \vee x_3'$	0
		5	$x_1' \vee x_2' \vee x_3'$	$x_1x_2x_3$	$x_1'x_3 \vee x_2x_3' \vee x_1x_2'$
		6	$x_1x_2x_3$	$x_1'x_2'x_3'$	0
	*	7	$x_1'x_2'x_3'$	$x_1x_2x_3$	$x_1'x_2'x_3'$

- Four choices: nodes 1, 2, 3, and a new gate
- Add a new gate: $ON_8 = OFF_8 = \varnothing$
- Cover m, connect gate, and propagate updates



Search Tree



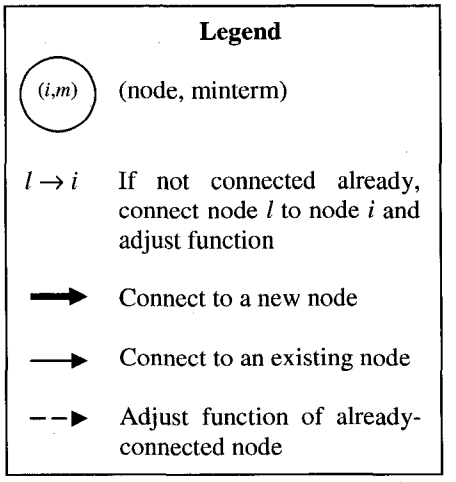


Figure 3.15: Trace and Search Tree for an infinite chain of gates that results when a new gate is repeatedly chosen for covering.

The order in which minterms are selected for covering can also lead to an infinite chain of gate nodes. The following trace shows how this type of chain occurs.

In this trace the 3-input XOR function is synthesized. The trace begins with an initial partial network containing 3 primary input nodes and a single output node. At each branching decision (decisions 3, 6, ...) a minterm from the node second from the bottom is chosen for covering. It is then covered by its one input node. The result of this covering will always be two structural implications to add two new gates to the end of the chain.

Decisions 1 and 2 in this trace are structural implications since both minterms selected for covering can only be covered by adding a new gate node to the network. In decision 3, the minterm $x_1 x_2' x_3$ from node 5 is chosen for covering. It is covered by the existing gate node, 6, which already exists as a fan-in to node 5. Following this decision two more structural implications are performed since both of the selected minterms can only be covered by adding a new gate to the network. Decision 6 again selects an uncovered minterm, $x_1 x_2' x_3$, from a node second from the bottom of the chain, node 7. This minterm is also covered by an existing gate node, 8, which already exists as a fan-in to node 7.

It is possible that this same type of decision followed by structural implications could continue to be made. If the cost bound for the network is infinite then the network will continue to grow without bound.

The third rule prevents this type of chaining by requiring that a minterm should be selected from a node whose fan-in cannot be used to cover the minterm. Using this rule, the chain of gates shown in this trace would be prevented at decision 3 where the uncovered minterm from node 6 would be chosen for covering rather than the node from 5.

Algorithm Trace: Infinite Chain of Gates (version 2)

INITIALIZATION

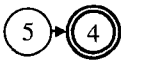
Struct	Conn	Node	ON_l	OFF_l	$UnCovered_l$
		1	x_1'	x_1	–
		2	x_2'	x_2	–
		3	x_3'	x_3	–
		4	$x_1 \oplus x_2 \oplus x_3$	$(x_1 \oplus x_2 \oplus x_3)'$	$x_1'x_2'x_3 \vee x_1'x_2x_3' \vee x_1x_2'x_3' \vee x_1x_2x_3$

1
2
3

Decision 1 Cover $(4, x_1'x_2'x_3)$: $i = 4$, $m = x_1'x_2'x_3$

Struct	Conn	Node	ON_l	OFF_l	$UnCovered_l$
S		1	x_1'	x_1	–
S		2	x_2'	x_2	–
S		3	x_3'	x_3	–
	*	4	$x_1 \oplus x_2 \oplus x_3$	$(x_1 \oplus x_2 \oplus x_3)'$	$x_1'x_2'x_3 \vee x_1'x_2x_3' \vee x_1x_2'x_3' \vee x_1x_2x_3$
• Add new gate 5 (Structural Implication) • Cover *m*, connect gate and propagate updates					

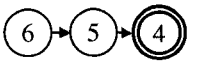
1
2
3



Decision 2 Cover $(5, x_1'x_2x_3)$: $i = 5$, $m = x_1'x_2x_3$

Struct	Conn	Node	ON_l	OFF_l	$UnCovered_l$
S		1	x_1'	x_1	–
S		2	x_2'	x_2	–
S		3	x_3'	x_3	–
		4	$x_1 \oplus x_2 \oplus x_3$	$(x_1 \oplus x_2 \oplus x_3)'$	$x_1'x_2x_3' \vee x_1x_2'x_3' \vee x_1x_2x_3$
	*	5	$(x_1 \oplus x_2 \oplus x_3)'$	$x_1'x_2'x_3$	$x_1'x_2'x_3' \vee x_1'x_2x_3 \vee x_1x_2'x_3 \vee x_1x_2x_3'$
• Add new gate 6 (Structural Implication) • Cover *m*, connect gate and propagate updates					

1
2
3

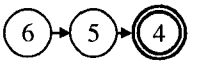


Decision 3 Cover $(5,\ x_1x_2'x_3)$: $i = 5$, $m = x_1x_2'x_3$

Struct	Conn	Node	ON_i	OFF_i	$UnCovered_i$
S		1	x_1'	x_1	–
S		2	x_2'	x_2	–
S		3	x_3'	x_3	–
		4	$x_1 \oplus x_2 \oplus x_3$	$(x_1 \oplus x_2 \oplus x_3)'$	$x_1'x_2x_3' \vee x_1x_2'x_3' \vee x_1x_2x_3$
	*	5	$(x_1 \oplus x_2 \oplus x_3)'$	$x_1'x_2'x_3$	$x_1'x_2'x_3' \vee x_1x_2'x_3 \vee x_1x_2x_3'$
S	C	6	$x_1'x_2'x_3$	$x_1'x_2x_3$	$x_1'x_2'x_3$

- Two choices: node 6 and a new gate
- Use node 6
- Cover m with 6 and propagate updates

1
2
3

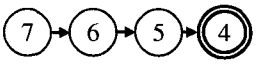


Decision 4 Cover $(6,\ x_1'x_2'x_3)$: $i = 6$, $m = x_1'x_2'x_3$

Struct	Conn	Node	ON_i	OFF_i	$UnCovered_i$
S		1	x_1'	x_1	–
S		2	x_2'	x_2	–
S		3	x_3'	x_3	–
		4	$x_1 \oplus x_2 \oplus x_3$	$(x_1 \oplus x_2 \oplus x_3)'$	$x_1'x_2x_3' \vee x_1x_2'x_3' \vee x_1x_2x_3$
		5	$(x_1 \oplus x_2 \oplus x_3)'$	$x_1'x_2'x_3$	$x_1'x_2'x_3' \vee x_1x_2x_3'$
	*	6	$x_1'x_2'x_3$	$x_1'x_2x_3 \vee x_1x_2'x_3$	$x_1'x_2'x_3$

- Add new gate 7 (Stuctural Implication)
- Cover m, connect gate and propagate updates

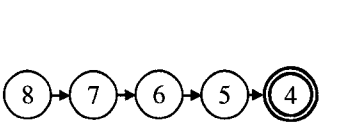
1
2
3



Decision 5 Cover $(7,\ x_1'x_2x_3)$: $i = 7, m = x_1'x_2x_3$

Struct	Conn	Node	ON_l	OFF_l	$UnCovered_l$
S		1	x_1'	x_1	–
S		2	x_2'	x_2	–
S		3	x_3'	x_3	–
		4	$x_1 \oplus x_2 \oplus x_3$	$(x_1 \oplus x_2 \oplus x_3)'$	$x_1'x_2x_3' \vee x_1x_2'x_3' \vee x_1x_2x_3$
		5	$(x_1 \oplus x_2 \oplus x_3)'$	$x_1'x_2'x_3$	$x_1'x_2'x_3' \vee x_1x_2x_3'$
		6	$x_1'x_2'x_3$	$x_1'x_2x_3 \vee x_1x_2'x_3$	0
	*	7	$x_1'x_2x_3 \vee x_1x_2'x_3$	$x_1'x_2'x_3$	$x_1'x_2x_3 \vee x_1x_2'x_3$

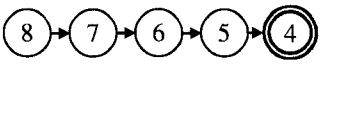
- Add new gate 8 (Structural Implication)
- Cover m, connect gate and propagate updates



Decision 6 Cover $(7,\ x_1x_2'x_3)$: $i = 7, m = x_1x_2'x_3$

Struct	Conn	Node	ON_l	OFF_l	$UnCovered_l$
S		1	x_1'	x_1	–
S		2	x_2'	x_2	–
S		3	x_3'	x_3	–
		4	$x_1 \oplus x_2 \oplus x_3$	$(x_1 \oplus x_2 \oplus x_3)'$	$x_1'x_2x_3' \vee x_1x_2'x_3' \vee x_1x_2x_3$
		5	$(x_1 \oplus x_2 \oplus x_3)'$	$x_1'x_2'x_3$	$x_1'x_2'x_3' \vee x_1x_2x_3'$
		6	$x_1'x_2'x_3$	$x_1'x_2x_3 \vee x_1x_2'x_3$	0
	*	7	$x_1'x_2x_3 \vee x_1x_2'x_3$	$x_1'x_2'x_3$	$x_1x_2'x_3$
S	C	8	$x_1'x_2'x_3$	$x_1'x_2x_3$	$x_1'x_2'x_3$

- Two choices: node 8 and a new gate
  Use node 8
- Cover m and propagate updates



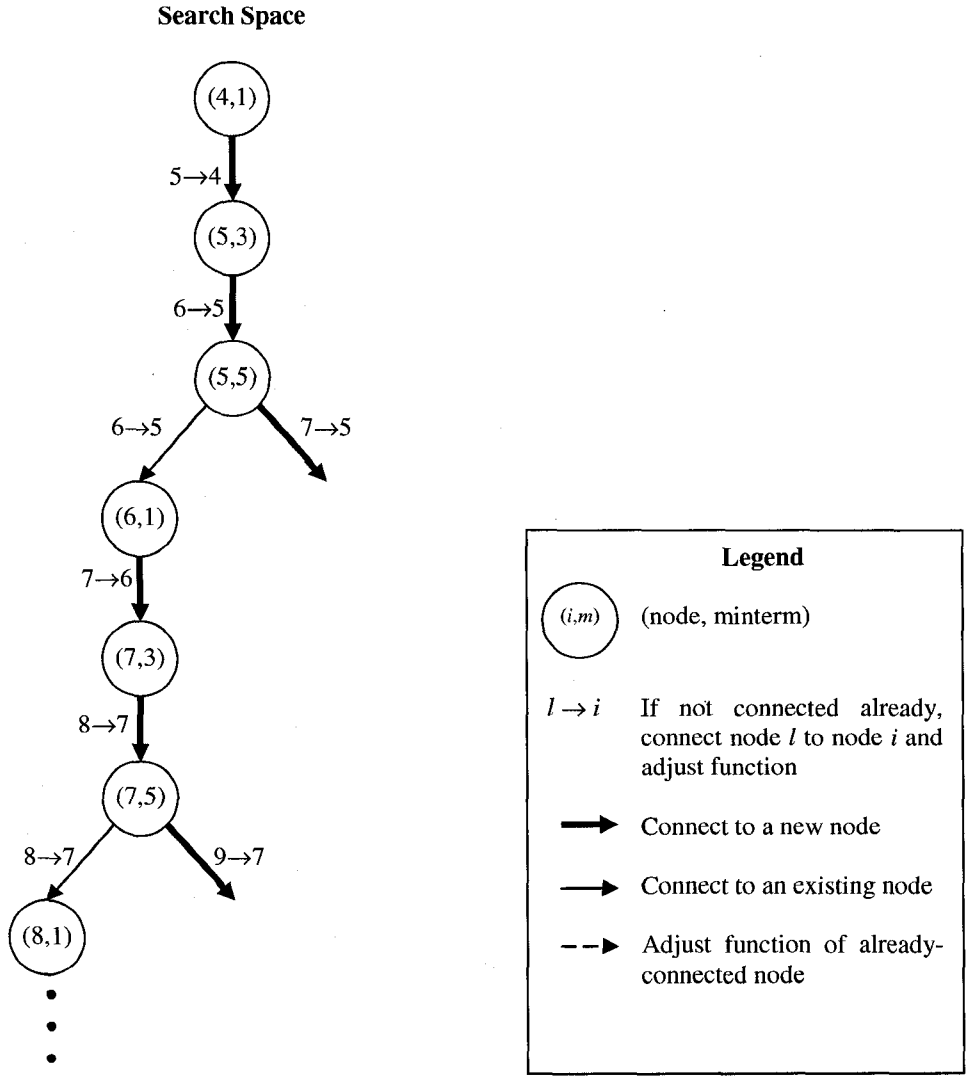


Figure 3.16: Trace and Search Tree for an infinite chain of gates that results when a poor selection of (node, minterm) pair is chosen for covering.

Adding these three techniques to BESS guarantees that the first path in the search tree will lead to a complete network after a finite number of recursive calls even if the cost bound is infinite. Once a complete network is found, the cost bound will be set to a finite value, and the remaining paths in the search tree will be finite because of this bound. Therefore the search tree will be finite.

3.8 Completeness

In addition to the property of convergence, a proof of completeness of a search algorithm is necessary to guarantee that the search will produce an optimal solution. BESS is complete if the search tree produced by the algorithm on a specific instance contains all optimal networks for this instance. This will then guarantee that the network found by the BESS is optimal.

Before this proof is given, some additional notation must be introduced. The portion of the search tree produced by the algorithm whose root node is represented by the network R is denoted by $\boldsymbol{T(R)}$. Therefore, if I is the initial network for a problem instance then $T(I)$ describes the entire search tree. To simplify the proof, we will assume that the algorithm does not prune based on the cost of the partial network. This pruning will be added in once completeness under these simpler assumptions have been proven. Note that when cost-based pruning is not used, each leaf of the search tree represents an elementary network.

The following lemmas and theorem provide the proof of completeness for BESS. The first lemma establishes the relationship between partial and complete networks when a covering is performed. The second lemma uses the first to establish a relationship between the search tree under the partial network and the complete network. Finally, the theorem pulls together the proof of completeness.

Lemma 3.1 *If a partial network P is a proper subnetwork of a complete network S then for any uncovered minterm m in P there exists at least one possible way of covering m such that the resulting network is also a subnetwork of S.*

Proof: If m is an uncovered minterm in P then m must belong to the on-set of some node i_P in P. Since P is a subnetwork of S, m must also belong to the on-set of the corresponding node i_S in S. In addition, the minterm m must be covered in i_S since S is a complete network. Let l_S be the node which covers m in S. Since P is a subnetwork of S there may or may not be a node in P that corresponds to this node l_S.

If there does not exist a node in P that corresponds to l_S, then i_P must have less than two nodes in its fan-in set. This is the case since the edge set of P must be contained in the edge set of S and the fan-in restriction on the gate nodes is 2. Since i_P does not have two inputs, the uncovered minterm m can be covered by adding a new gate node l_P to P along with the edge (l_P, i_P). Based on the covering and functional implication rules, the function at l_P will be $\mathrm{ON}_{l_P} = \mathrm{OFF}_{i_P}$ and $\mathrm{OFF}_{l_P} = m$. Thus, $\mathrm{ON}_{l_P} \leq \mathrm{ON}_{l_S}$ since $\mathrm{OFF}_{i_P} \leq \mathrm{OFF}_{i_S}$ and $\mathrm{ON}_{l_S} = \mathrm{OFF}_{i_S}$. Simiarly, $\mathrm{OFF}_{l_P} \leq \mathrm{OFF}_{l_S}$ since l_S covers the minterm m. The resulting network will be a subnetwork of S since the added gate node and edge correspond to an existing gate node and edge in S and the function at l_P will be a subset of the function at l_S.

If there already exists a node l_P in P that corresponds to l_S then l_P must be connectible to i_P and $m \wedge \mathrm{ON}_{l_P} = 0$. This must be the case since P is a subnetwork of S. Therefore l_P can be used to cover m by adding the edge (l_P, i_P) to the network P (if it is not already there) and adding m to the off-set of l_P (if it is not already there). The resulting network will be a subnetwork of S since the added edge in P corresponds to an existing edge in S and l_S covers the minterm m in i_S so $\mathrm{OFF}_{l_P} \leq \mathrm{OFF}_{l_S}$. □

Lemma 3.2 *Let S be an elementary network and let P be a subnetwork of S. Any search tree* $T(P)$ *will contain the elementary network S as one of its leaf nodes.* [Nakagawa 89]

Proof: Suppose the search tree $T(P)$ does not contain the network S as one of its leaf nodes. In this search tree, there must exist a network $\hat{P}$ that is a subnetwork of S but represents a node in the search tree such that the only network in $T(\hat{P})$ that is a subnetwork of S is $\hat{P}$. Since $\hat{P}$ is a subnetwork of S, two cases are possible: either $\hat{P} = S$ or $\hat{P}$ is a proper subnetwork of S. If $\hat{P} = S$ then this contradicts the assumption that S is not contained in the search tree $T(P)$. Therefore $\hat{P}$ is a proper subnetwork of S. Since S is an elementary network, this implies that $\hat{P}$ is not complete. Let m be an uncovered minterm in $\hat{P}$. Since $\hat{P}$ is a proper subnetwork of S, then by Lemma 3.1 there is at least one way to cover the minterm m such that the resulting network will also be a subnetwork of S. Therefore at least one network representing a child of $\hat{P}$ in the search tree $T(\hat{P})$ must be a subnetwork of S. This too is a contradiction however as $\hat{P}$ is the only network in $T(\hat{P})$ that is a subnetwork of S. □

Theorem 3.1. *A search tree T produced by BESS will contain all optimal networks.*

Proof: If Lemma 3.2 is applied to the case where I is the initial network then the search tree $T(I)$ will contain every elementary network. By definition, every optimal network is an elementary network, this implies that the search tree will contain every optimal network. □

Theorem 3.1 proves that a version of BESS which does not use pruning is complete. The pruning part of the algorithm will only remove sections of the search tree which have networks with cost greater than the current cost bound. The cost bound is set when a complete network is reached; therefore the cost bound will never be less than the cost of the optimal networks. This implies that the version of BESS which uses pruning based on the cost bound will only remove sections of the search tree which have networks with cost greater than the cost of the optimal networks. Therefore an optimal network will be produced by BESS when pruning based on cost is used.

3.9 Pruning

Using BESS, it is often the case that a specific partial networks will be generated more than once at different points in the search. One reason for this regeneration is the fact that both inputs to a node may cover the same minterm (*overlapped covering*). Another reason comes from the symmetry that is present in the network: either symmetry of the NAND gate with respect to its inputs (*local symmetry*) or the symmetry of the output functions with respect to the primary inputs (*global symmetry*). By being of aware of the possibilities that repetitive networks may exist and knowing how they can be created, pruning

techniques can be added to the algorithm to check for these situations and prune repetitive portions of the search tree.

3.9.1 Pruning Based on Overlapped Covering

The definition of covering from Section 2.6 stated that a minterm contained in the on-set of a node (i,j,k) is covered if the minterm appears in the off-set of **either** input. This definition leaves the possibility that the minterm will appear in the off-set of both of the nodes j and k. In this case both nodes will cover the minterm. This overlap in covering may lead to the same network being generated in two separate steps of the synthesis algorithm.

To demonstrate how this overlapped covering can result in a regeneration of a network, we provide a portion of a trace on the following three pages. This trace shows the synthesis of the 4-input function $x_1x_2 \vee x_3x_4$. In Decision 2, the minterm $x_1x_2x_3x_4$ from node 5 is selected for covering. There are two options for covering this minterm, the existing input node 6 can be used or a new gate node can be added to the network to cover the minterm.

When node 6 is used to cover, decisions 3 through 7 follow producing the complete network shown in Decision 7. When a new node is used in the covering, decisions 8 through 11 follow producing the complete network shown in Decision 11. A comparison of the two complete networks shows that both paths eventually lead to the same complete network. Therefore we can conclude that one of these paths is redundant.

This repetition results from an overlap in the covering of the minterm $x_1x_2x_3x_4$ in node 5 at Decision 2. If the algorithm did not allow node 6 to cover the minterm $x_1x_2x_3x_4$ in any network created from the second branch (Decisions 8 through 11), then the duplicate complete network would never be created, thus pruning unnecessary portions of this search tree.

Algorithm Trace: Repetitive Network Example based on Overlapped Covering

INITIALIZATION

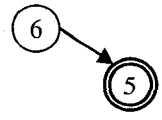
Node	ON_l	OFF_l	$UnCovered_l$
5	$x_1x_2 \vee x_3x_4$	$(x_1' \vee x_2')(x_3' \vee x_4')$	$x_1x_2 \vee x_3x_4$

1 2 3 4

Decision 1 Cover (5, 12): $i = 5, m = 12$

Node	ON_l	OFF_l	$UnCovered_l$
5	$x_1x_2 \vee x_3x_4$	$(x_1' \vee x_2')(x_3' \vee x_4')$	$x_1x_2 \vee x_3x_4$
•Add new gate 6 (Structural Implication) •Cover *m*, connect gate and propagate updates			

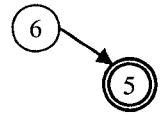
1 2 3 4



Decision 2 Cover (5, 15): $i = 5, m = 15$

Node	ON_l	OFF_l	$UnCovered_l$
5	$x_1x_2 \vee x_3x_4$	$(x_1' \vee x_2')(x_3' \vee x_4')$	$x_1x_2x_3 \vee x_1x_2x_4 \vee x_3x_4$
6	$(x_1' \vee x_2')(x_3' \vee x_4')$	$x_1x_2x_3'x_4'$	$(x_1' \vee x_2')(x_3' \vee x_4')$
•Two choices: node 6 and a new gate •Cover *m* with node 6 and propagate updates			

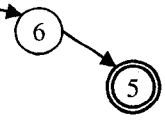
1 2 3 4



Decision 3 Cover (6, 4): $i = 6, m = 4$

Node	ON_l	OFF_l	$UnCovered_l$
5	$x_1x_2 \vee x_3x_4$	$(x_1' \vee x_2')(x_3' \vee x_4')$	$x_1'x_3x_4 \vee x_2'x_3x_4 \vee x_1x_2x_3'x_4 \vee x_1x_2x_3x_4'$
6	$(x_1' \vee x_2')(x_3' \vee x_4')$	$x_1x_2x_3'x_4' \vee x_1x_2x_3x_4$	$(x_1' \vee x_2')(x_3' \vee x_4')$
•Two choices: node 1 and a new gate • Connect node 1 and propagate updates			

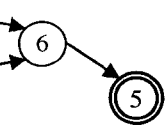
1 2 3 4



Decision 4 Cover (6, $x_1x_2'x_3'x_4'$): $i = 6, m = x_1x_2'x_3'x_4'$

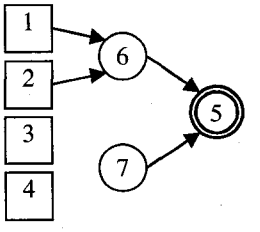
Node	ON_l	OFF_l	$UnCovered_l$
5	$x_1x_2 \vee x_3x_4$	$(x_1' \vee x_2')(x_3' \vee x_4')$	$x_1'x_3x_4 \vee x_2'x_3x_4 \vee x_1x_2x_3'x_4 \vee x_1x_2x_3x_4'$
6	$x_1' \vee x_2'x_3' \vee x_2'x_4'$	$x_1x_2x_3'x_4' \vee x_1x_2x_3x_4$	$x_1x_2'x_3' \vee x_1x_2'x_4'$
•Two choices: node 2 and a new gate •Connect node 2 and propagate updates			

1 2 3 4



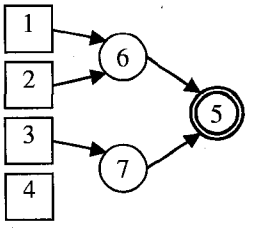
Decision 5 Cover (5, $x_1'x_2'x_3x_4$): $i = 5, m = x_1'x_2'x_3x_4$

Node	ON_i	OFF_i	$UnCovered_i$
5	$x_1x_2 \vee x_3x_4$	$(x_1' \vee x_2')(x_3' \vee x_4')$	$x_1'x_3x_4 \vee x_2'x_3x_4$
6	$x_1' \vee x_2'$	x_1x_2	0
•Add new gate 7 (Structural Implication) •Cover m, connect gate and propagate updates			



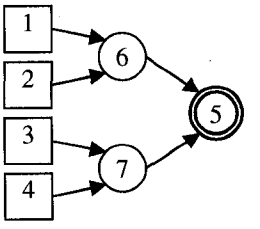
Decision 6 Cover (7, $x_1'x_2'x_3'x_4$): $i = 7, m = x_1'x_2'x_3'x_4$

Node	ON_i	OFF_i	$UnCovered_i$
5	$x_1x_2 \vee x_3x_4$	$(x_1' \vee x_2')(x_3' \vee x_4')$	0
6	$x_1' \vee x_2'$	x_1x_2	0
7	$(x_1' \vee x_2')(x_3' \vee x_4')$	$x_1'x_3x_4 \vee x_2'x_3x_4$	$(x_1' \vee x_2')(x_3' \vee x_4')$
•Two choices: node 3 and a new gate •Connect node 3 and propagate updates			



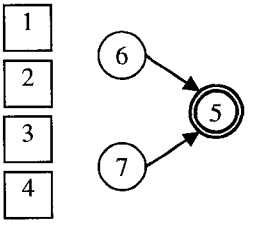
Decision 7 Cover (7, $x_1'x_2'x_3x_4'$): $i = 7, m = x_1'x_2'x_3x_4'$

Node	ON_i	OFF_i	$UnCovered_i$
5	$x_1x_2 \vee x_3x_4$	$(x_1' \vee x_2')(x_3' \vee x_4')$	0
6	$x_1' \vee x_2'$	x_1x_2	0
7	$x_3' \vee x_1'x_4' \vee x_2'x_4'$	$x_1'x_3x_4 \vee x_2'x_3x_4$	$x_1'x_3x_4' \vee x_2'x_3x_4'$
•Two choices: node 4 and a new gate •Connect node 4 and propagate updates			
5	$x_1x_2 \vee x_3x_4$	$(x_1' \vee x_2')(x_3' \vee x_4')$	0
6	$x_1' \vee x_2'$	x_1x_2	0
7	$x_3' \vee x_4'$	x_3x_4	0



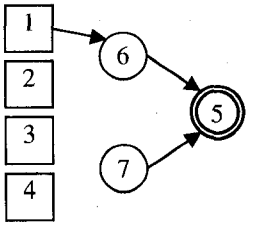
Decision 2 Cover (5, $x_1x_2x_3x_4$): $i = 5, m = x_1x_2x_3x_4$

Node	ON_i	OFF_i	$UnCovered_i$
5	$x_1x_2 \vee x_3x_4$	$(x_1' \vee x_2')(x_3' \vee x_4')$	$x_1x_2x_3 \vee x_1x_2x_4 \vee x_3x_4$
6	$(x_1' \vee x_2')(x_3' \vee x_4')$	$x_1x_2x_3'x_4'$	$(x_1' \vee x_2')(x_3' \vee x_4')$
•Two choices: node 6 and a new gate •Connect new gate 7, cover m, and propagate updates			



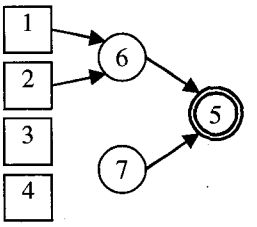
Decision 8 Cover (6, $x_1'x_2x_3'x_4'$): $i = 6$, $m = x_1'x_2x_3'x_4'$

Node	ON_i	OFF_i	$UnCovered_i$
5	$x_1x_2 \vee x_3x_4$	$(x_1' \vee x_2')(x_3' \vee x_4')$	$x_1'x_3x_4 \vee x_2'x_3x_4 \vee x_1x_2x_3'x_4 \vee x_1x_2'x_3x_4'$
6	$(x_1' \vee x_2')(x_3' \vee x_4')$	$x_1x_2x_3'x_4'$	$(x_1' \vee x_2')(x_3' \vee x_4')$
7	$(x_1' \vee x_2')(x_3' \vee x_4')$	$x_1x_2x_3x_4$	$(x_1' \vee x_2')(x_3' \vee x_4')$
•Two choices: node 1 and a new gate •Connect 1 and propagate updates			



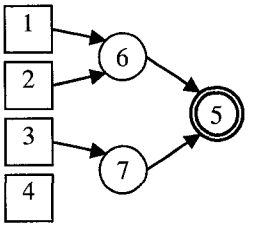
Decision 9 Cover (6, $x_1x_2'x_3'x_4'$): $i = 6$, $m = x_1x_2'x_3'x_4'$

Node	ON_i	OFF_i	$UnCovered_i$
5	$x_1x_2 \vee x_3x_4$	$(x_1' \vee x_2')(x_3' \vee x_4')$	$x_1x_2'x_3x_4 \vee x_1x_2x_3'x_4 \vee x_1x_2x_3x_4'$
6	$x_1' \vee x_2'x_3' \vee x_2'x_4'$	$x_1x_2x_3'x_4'$	$x_1x_2'x_3' \vee x_1x_2'x_4'$
7	$(x_1' \vee x_2')(x_3' \vee x_4')$	$x_1'x_3x_4 \vee x_2x_3x_4$	$(x_1' \vee x_2')(x_3' \vee x_4')$
•Two choices: node 2 and a new gate •Connect 2 and propagate updates			



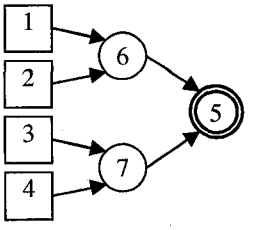
Decision 10 Cover (7, $x_1'x_2'x_3'x_4$): $i = 7$, $m = x_1'x_2'x_3'x_4$

Node	ON_i	OFF_i	$UnCovered_i$
5	$x_1x_2 \vee x_3x_4$	$(x_1' \vee x_2')(x_3' \vee x_4')$	0
6	$x_1' \vee x_2'$	x_1x_2	0
7	$(x_1' \vee x_2')(x_3' \vee x_4')$	x_3x_4	$(x_1' \vee x_2')(x_3' \vee x_4')$
•Two choices: node 3 and a new gate •Connect 3 and propagate updates			



Decision 11 Cover (7, $x_1'x_2'x_3x_4'$): $i = 7$, $m = x_1'x_2'x_3x_4'$

Node	ON_i	OFF_i	$UnCovered_i$
5	$x_1x_2 \vee x_3x_4$	$(x_1' \vee x_2')(x_3' \vee x_4')$	0
6	$x_1' \vee x_2'$	x_1x_2	0
7	$x_3' \vee x_1'x_4' \vee x_2'x_4'$	x_3x_4	$x_1'x_2'x_3x_4' \vee x_1'x_2x_3x_4' \vee x_1x_2'x_3x_4' \vee$
•Two choices: node 4 and a new gate •Connect 4 and propagate updates			
5	$x_1x_2 \vee x_3x_4$	$(x_1' \vee x_2')(x_3' \vee x_4')$	0
6	$x_1' \vee x_2'$	x_1x_2	0
7	$x_3' \vee x_4'$	x_3x_4	0



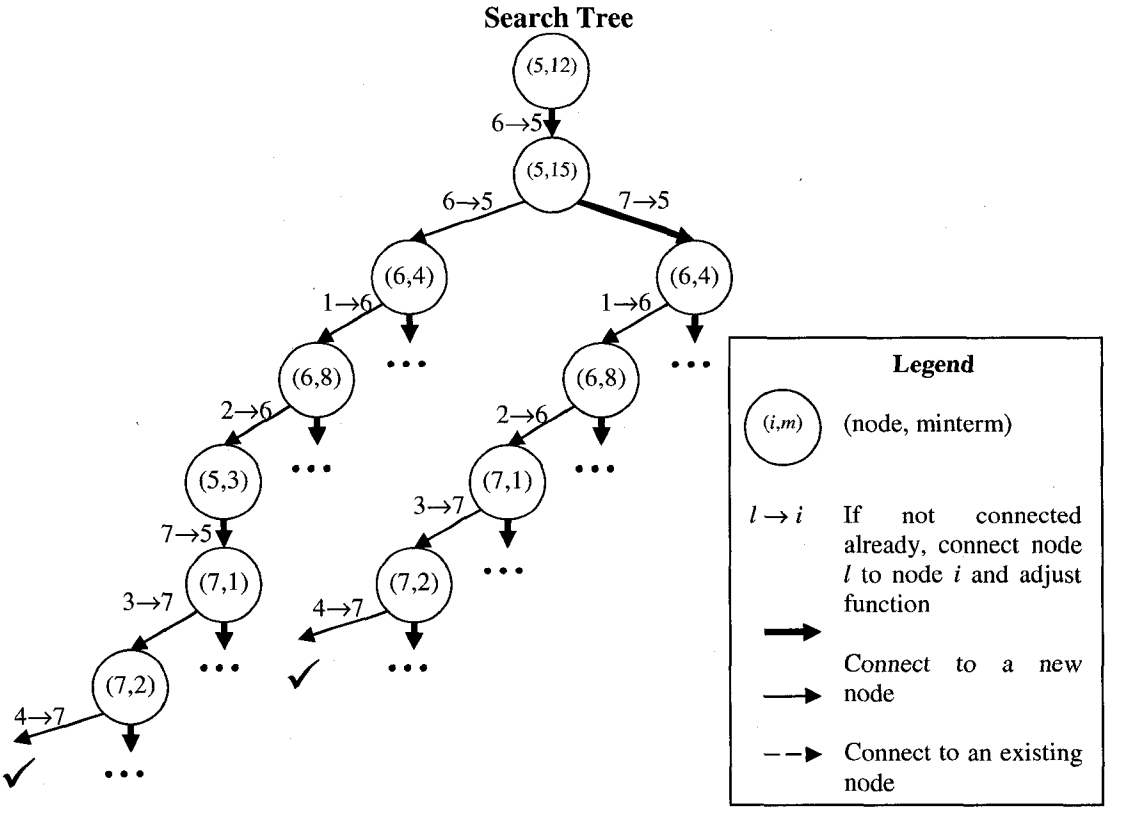


Figure 3.17: Trace and Search Tree for an overlapped covering example

In general, the only time this type of pruning can be done is after a fan-in node has been used to cover a minterm. If a node l in the connectible set is a fan-in of the node (i, j, k) and l is used to cover the minterm m then for the remaining networks created by the alternate coverings of m, m can be added to the on-set of the node l. The result will be that the every network contained in the portion of the search space represented by each of these new networks will not have the node l covering m. This in turn prevents a network from the space where l covers m to be repeated.

An updated version of the SynthesizeNetwork procedure is given in Figure 3.18 to include this pruning technique. Once a node l is used to cover the minterm m, if l is an input of (i, j, k), the minterm m can be added to the on-set of l ensuring that l will not be able cover m for the rest of the partial networks created from this call.

```
SynthesizeNetwork v.6 (Network)
  if ( ∧ (i∈ gate nodes) UnCovered_i = 0 )
     if (number gates in Network is smallest seen so far)
        Store Network as current minimum
        UpperBound ← Cost_Network
  else
     m, (i,j,k) ← SelectMintermForCovering
     CovNodes ← FindAllCoveringNodes(m, (i,j,k))
     for (all l ∈ CovNodes)
        NewNetwork ← PerformCovering((i,j,k), l, m)
        PropagateFunctionalImplications((i,j,k))
        SimpleGlobalFtnImpl(NewNetwork)
        GeneralGlobalFtnImpl(NewNetwork)
        do
           UpdateConnectibleSet()
           AddNewGate (NewNetwork)
        while ( nodes were added to NewNetwork)
        if (Cost_NewNetwork < UpperBound)
           SynthesizeNetwork(NewNetwork)
        if (l ∉ {j,k})
           ON_l := ON_l ∨ m
```

Figure 3.18: SynthesizeNetwork Procedure –Version 6 - Pruning based on Overlapped Covering

In order for this pruning technique to be useful, it must maintain the completeness of BESS. The search tree produced by the algorithm must still contain all optimal solutions.

Lemma 3.3 *Given a partial network P, let m be an uncovered minterm of some gate (i, j, k) in P where j is an input of the node that can cover the minterm m. Let P_1 and P_0 be the partial networks where P_1 has m in the on-set of j and P_0 has m in the off-set of j. All elementary networks in the search tree $T(P)$ are in $T(P_1) \cup T(P_0)$.* [Nakagawa 89]

Proof: For every elementary network S in $T(P)$, P is a subnetwork of S. Therefore at least one of P_1 or P_0 is a subnetwork of S. By applying Lemma 3.2 to the cases when $P = P_1$ and $P = P_0$ every elementary network contained in $T(P)$ will also be contained in either $T(P_1)$ or $T(P_0)$. □

Theorem 3.2. *A version of BESS using the pruning technique based on overlapped covering will generate every optimal solution.* [Nakagawa 89]

Proof: Let P be a partial network in the search tree of the algorithm and let $R_1, \ldots, R_k$ be the children of P in the search tree. These networks are the networks created by the alternate ways of covering an uncovered minterm m from P. If the pruning technique was not used to in this step, then Lemma 3.2 states thar every optimal solution contained in $T(P)$ is contained in one of $T(R_1), \ldots, T(R_k)$.

If the pruning technique was used in this step then $R_1 = P_0$ from Lemma 3.3 and P_1 from Lemma 3.3 will be a subnetwork of each of the networks $R_2, \ldots, R_k$ and this lemma implies that every optimal

network is either in R_1 or P_1. By using Lemma 3.1 at least one of the search trees $T(R_2),\ldots,T(R_k)$ will contain each optimal network described by P_1. Therefore no optimal networks in the search tree $T(P)$ will be lost. □

3.9.2 Pruning based on Local Symmetry

The NAND function is symmetric in its inputs implying that the NAND function on the inputs x and y (i.e. $f(x, y) = x' \vee y'$) produces the same result no matter the order of x and y, $f(x, y) = f(y, x)$. Therefore the order in which two nodes are connected to a NAND gate does not matter in terms of the resulting function. The two nodes (i, j, k) and (i, k, j) in Figure 3.19 will have the same Boolean function.

Figure 3.19: Symmetry of the NAND gate

Duplicate networks based on this type of symmetry are generated when two nodes l_1 and l_2 can be used to cover a single minterm in a gate node (i, j, k). When the minterm m is selected for covering, two networks are created: P_1 is the network which adds the edge (l_1, i) to cover m and P_2 is the network which adds the edge (l_2, i) to cover m. In subsequent calls on these networks a minterm from node (i, j, k) may again be selected for covering and the opposite node used to cover this minterm. (i.e. the edge (l_2, i) is added in P_1 and the edge (l_1, i) is added in P_2) Thus producing identical networks except for the order in which the fan-in nodes appear in the node (i ,j, k). The goal of symmetric pruning is to prohibit the second duplicate partial network from being produced while not eliminating any elementary networks from the search space.

The trace that follows illustrates this type of repetition and suggests a potential solution. In this trace, the two-input AND function is synthesized. The entire search of the algorithm is shown in the trace.

At decision 2, the minterm $x_1'x_2'$ from node 4 is selected for covering. There are three possible ways to cover this minterm which implies that three partial networks are created. Each of these coverings is shown in the trace. The first, labeled Decision 2, uses node 1 to cover the minterm. The second, labeled Decision 2b, uses node 2 to cover the minterm. The third, labeled Decision 2c, uses a new gate node, node 5, to cover the minterm. When the SynthesizeNetwork procedure is called from each of these networks, a minterm from node 4 is selected for covering. These covering steps are given in Decisions 3, 4, and 5. Due to the symmetry of the NAND gate, duplicate networks will result from this covering at these decision points. The networks created from Decisions 3 and 4 are the same while the networks that result from Decisions 3b and 5 are the same.

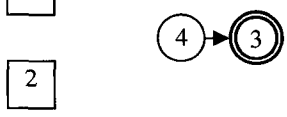
Algorithm Trace: Repetitive Network Example based on Local Symmetry

INITIALIZATION

Node l	ON_l	OFF_l	$UnCovered_l$
3	x_1x_2	$x_1' \vee x_2'$	x_1x_2

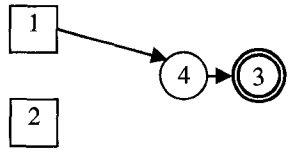
Decision 1 Cover (3, x_1x_2): $i = 3$, $m = x_1x_2$

Node l	ON_l	OFF_l	$UnCovered_l$
3	x_1x_2	$x_1' \vee x_2'$	x_1x_2
• Add new gate: node 4 (Structural Implication) • Cover m, connect gate and propagate updates			



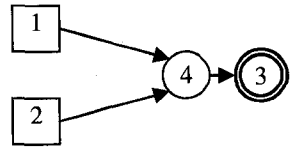
Decision 2 Cover (4, $x_1'x_2'$): $i = 4$, $m = x_1'x_2'$

Node l	ON_l	OFF_l	$UnCovered_l$
3	x_1x_2	$x_1' \vee x_2'$	0
4	$x_1' \vee x_2'$	x_1x_2	$x_1' \vee x_2'$
• Three choices: nodes 1, 2, and a new gate. • Connect node 1 and propagate updates			



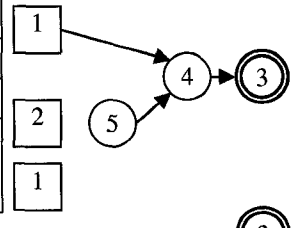
Decision 3 Cover (4, x_1x_2'): $i = 4$, $m = x_1x_2'$

Node l	ON_l	OFF_l	$UnCovered_l$
3	x_1x_2	$x_1' \vee x_2'$	0
4	$x_1' \vee x_2'$	x_1x_2	x_1x_2'
• Two choices: node 2 and a new gate • Connect node 2 and propagate updates			



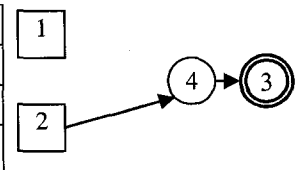
Decision 3(b) Cover (4, x_1x_2'): $i = 4$, $m = x_1x_2'$

Node l	ON_l	OFF_l	$UnCovered_l$
3	x_1x_2	$x_1' \vee x_2'$	0
4	$x_1' \vee x_2'$	x_1x_2	x_1x_2'
• Two choices: node 2 and a new gate • Connect a new gate, node 5, cover m, and propagate updates			



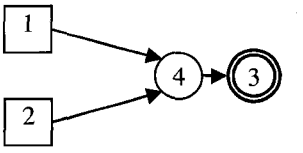
Decision 2(b) Cover (4, $x_1'x_2'$): $i = 4$, $m = x_1'x_2'$

Node l	ON_l	OFF_l	$UnCovered_l$
3	x_1x_2	$x_1' \vee x_2'$	0
4	$x_1' \vee x_2'$	x_1x_2	$x_1' \vee x_2'$
• Three choices: nodes 1, 2, and a new gate • Connect node 2 and propagate updates			



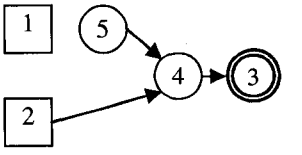
Decision 4 Cover (4, $x_1'x_2$): $i = 4, m = x_1'x_2$

Node l	ON_l	OFF_l	$UnCovered_l$
3	x_1x_2	$x_1' \vee x_2'$	0
4	$x_1' \vee x_2'$	x_1x_2	$x_1'x_2$
• Two choices: node1 and a new gate • Connect node 1 and propagate updates			



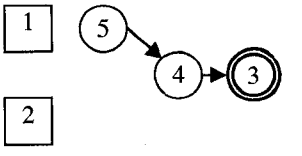
Decision 4(b) Cover (4, $x_1'x_2$): $i = 4, m = x_1'x_2$

Node l	ON_l	OFF_l	$UnCovered_l$
3	x_1x_2	$x_1' \vee x_2'$	0
4	$x_1' \vee x_2'$	x_1x_2	$x_1'x_2$
• Two choices: node1 and a new gate • Connect a new gate, node 5, cover m, and propagate updates			



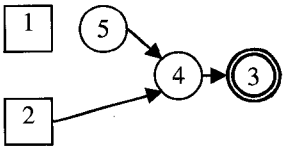
Decision 2(c) Cover (4, $x_1'x_2'$): $i = 4, m = x_1'x_2'$

Node l	ON_l	OFF_l	$UnCovered_l$
3	x_1x_2	$x_1' \vee x_2'$	0
4	$x_1' \vee x_2'$	x_1x_2	$x_1' \vee x_2'$
• Three choices: nodes 1, 2, and a new gate • Connect a new gate, node 5, cover m, and propagate updates			



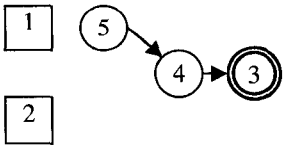
Decision 5 Cover (4, x_1x_2'): $i = 4, m = x_1x_2'$

Node l	ON_l	OFF_l	$UnCovered_l$
3	x_1x_2	$x_1' \vee x_2'$	0
4	$x_1' \vee x_2'$	x_1x_2	$x_1'x_2 \vee x_1x_2'$
5	x_1x_2	$x_1'x_2'$	x_1x_2
• Three choices: nodes 2, 5, and a new gate • Connect node 2 and propagate updates			



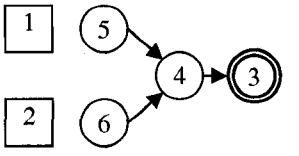
Decision 5(b) Cover (4, x_1x_2'): $i = 4, m = x_1x_2'$

Node l	ON_l	OFF_l	$UnCovered_l$
3	x_1x_2	$x_1' \vee x_2'$	0
4	$x_1' \vee x_2'$	x_1x_2	$x_1'x_2 \vee x_1x_2'$
5	x_1x_2	$x_1'x_2'$	x_1x_2
• Three choices: nodes 2, 5, and a new gate • Connect node 5, cover m, and propagate update			



Decision 5(c) Cover (4, x_1x_2'): $i = 4$, $m = x_1x_2'$

Node l	ON_l	OFF_l	$UnCovered_l$
3	x_1x_2	$x_1' \vee x_2'$	0
4	$x_1' \vee x_2'$	x_1x_2	$x_1'x_2 \vee x_1x_2'$
5	x_1x_2	$x_1'x_2'$	x_1x_2
• Three choices here: nodes 1, 5, and a new gate • Connect a new gate, node 6, cover m, and propagate updates			



Search Space

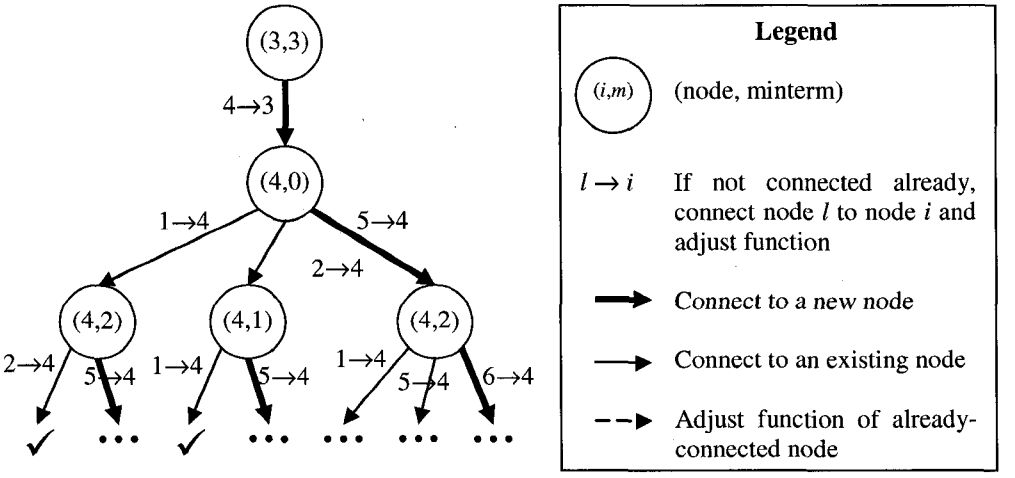


Figure 3.20: Trace and Search Tree for a Repetitive Network Example based on Local Symmetry

The algorithm will simply be repeating the search of an already explored subspace if the SynthesizeNetwork procedure is called on both networks in these pairs of equivalent partial networks. Instead, the algorithm should determine that the space represented by the partial network has already been searched and then prune the partial network based on that information.

To prune these duplicate partial networks, a *non-connectible set* can be added to the node structure. If a node l is in the *non-connectible set* of a node i then l cannot be used to cover anything in i. l should never appear in the connectible set of i. This non-connectible set can then be used to prevent partial networks from being repeated. If at some step in the algorithm nodes l_1 and l_2 can be used to cover a minterm in a gate node (i, j, k), then as before, two partial networks are created P_1 with the edge (l_1, i) added and P_2 with the edge (l_2, i) added. However, P_2 will also have l_1 added to the non-connectible set of i. In subsequent calls on these partial networks when a minterm from node i is again selected for covering, the edge (l_2, i) can be added in P_1, but the edge (l_1, i) cannot be added in P_2. Therefore the partial network that has l_1 and l_2 as the two inputs of node i will only occur once.

In the above trace, the non-connectible set of node 4 in decision 2(b) is the set {1}. Therefore, when the minterm $x_1'x_2$ from 4 is selected for covering in decision 4, the node x_1 cannot be used as one of the options

for performing this covering. The result would be that only the network from decision 4(b) will be generated. Similarly, the non-connectible set of node 4 in decision 2(c) would be the set {1,2}, so the network at decision 5 would not be generated.The SynthesizeNetwork procedure is changed to include this symmetric pruning. Once a node *l* is used to cover the minterm *m*, *l* is added to the non-connectible set of *i*. Therefore *l* will appear in the non-connectible set for the rest of the partial networks created in this step.

```
SynthesizeNetwork v.7 (Network)
  if ( ⋀_{i∈ gate nodes} UnCovered_i = 0 )
    if (number gates in Network is smallest seen so far)
      Store Network as current minimum
      UpperBound ← Cost_Network
  else
    m, (i,j,k) ← SelectMintermForCovering
    CovNodes ← FindAllCoveringNodes(m, (i,j,k))
    for (all l ∈ CovNodes)
      NewNetwork ← PerformCovering((i,j,k), l, m)
      PropagateFunctionalImplications((i,j,k))
      SimpleGlobalFtnImpl(NewNetwork)
      GeneralGlobalFtnImpl(NewNetwork)
      do
        UpdateConnectibleSet()
        AddNewGate (NewNetwork)
      while ( nodes were added to NewNetwork)
      if (Cost_NewNetwork < UpperBound)
        SynthesizeNetwork(NewNetwork)
      if (l ∉ {j,k})
        ON_l := ON_l ∨ m
      else if (m ∈ OFF_l)
        Add l to NonConnectible_i
```

Figure 3.21: SynthesizeNetwork Procedure – Version 7 - with Local Symmetry Pruning

In addition to the change in the SynthesizeNetwork procedure, the UpdateConnectibleSets procedure must also be changed. This procedure must ensure that any node contained in the connectible set of a node does not also appears in the node's non-connectible set.

Once again, this pruning technique can only be used in the synthesis algorithm if it maintains the completeness of BESS. This implies that the search tree of the algorithm must still contain all optimal solutions.

Lemma 3.4 *Let P be a partial network containing two nodes i and l. Let P_1 and P_0 be two supernetworks of P such that $T(P_1)$ and $T(P_0)$ form a partition of $T(P)$. $T(P_0)$ is the subset of the networks in $T(P)$ that do not contain the edge (l,i). $T(P_1)$ is the subset of the networks in $T(P)$ that contain the edge (l,i).* [Nakagawa 89]

Proof: For every optimal solution S in $T(P)$, P is a subnetwork of S. Therefore either the network P_1 which contains the edge (l,i) is subnetwork of S or the network P_0 which does not contain the edge (l,i) is a subnetwork of S. By applying Lemma 3.2 to the cases when $P = P_1$ and $P = P_0$ we have every optimal solution contained in $T(P)$ contained in either $T(P_0)$ or $T(P_1)$. □

Theorem 3.3. *A version of BESS using the pruning technique based on local symmetry will generate every optimal solution.* [Nakagawa 89]

Proof: Let P be a partial network in the search tree of the algorithm and let R_1, ..., R_k be the children of P in the search tree. These networks are the networks created by the alternate ways of covering an uncovered minterm m from P. If the pruning technique was not used to in this step, then Lemma 3.2 or Theorem 3.2 states every optimal solution contained in $T(P)$ is contained in one of $T(R_1),\ldots,T(R_k)$.

If the pruning technique was used in this step then $R_1 = P_1$ from Lemma 3.4 and P_0 from Lemma 3.4 will be a subnetwork of each of the networks R_2, ... , R_k. Lemma 3.4 implies that every optimal network is either in R_1 or P_0. By using Lemma 3.1 at least one of the search trees $T(R_2),\ldots,T(R_k)$ will contain each optimal network in P_0. Therefore no optimal networks in the search tree $T(P)$ will be lost. □

3.9.3 Pruning based on Global Symmetry

While there are many types of symmetry, we will only consider one other type that exists in some synthesis instances. In total symmetry, the function(s) remains invariant under all possible permutations of the variables. Such symmetry can be found within the three networks given in Figure 3.22. All three networks are implementations of the function $x_1' \vee x_2' \vee x_3'$. In addition, each network can be retrieved from the others by simply permuting the input nodes {1, 2, 3}.

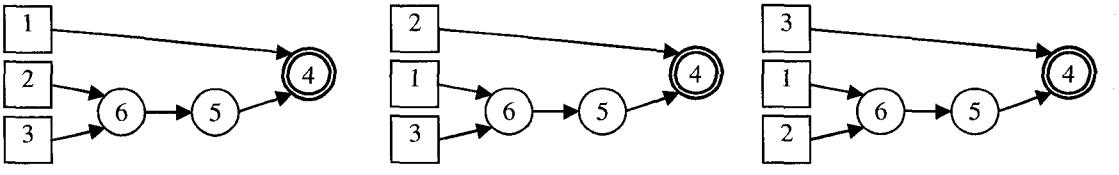


Figure 3.22: Duplicate Networks from Global Symmetry

This type of duplicate network exists for every totally symmetric function.

Lemma 3.5 *Let S and $\hat{S}$ be elementary networks such that $\hat{S}$ is simply S with the set of input nodes $(1,\ldots,n)$ permuted to $(h_1,\ldots,h_n)$. For any node i in S, if F_i is the global function at i and $\widehat{F}_i$ is the global function at the corresponding node i in $\hat{S}$, then $\widehat{F}_i(x_1,\ldots,x_n) = F_i(x_{h_1},\ldots,x_{h_n})$.*

Proof: (by induction)

The only nodes at level $n = 0$ are the input nodes. An input node i has global functions $F_i = x_i$ and $\widehat{F}_i = x_{h_i}$. Therefore $\widehat{F}_i(x_1,\ldots,x_n) = F_i(x_{h_1},\ldots,x_{h_n})$.

Assume for all nodes i with level less than n, $\widehat{F}_i(x_1,\ldots,x_n) = F_i(x_{h_1},\ldots,x_{h_n})$.

A gate node (i, j, k) in S at level k with two inputs j and k will have a global function $F_i = (F_j \wedge F_k)'$ while the corresponding node in $\hat{S}$ will have a global function $\widehat{F}_i = (\widehat{F}_j \wedge \widehat{F}_k)'$. The nodes j and k will have level less than n so by the induction assumption, $\widehat{F}_j(x_1,\ldots,x_n) = F_j(x_{h_1},\ldots,x_{h_n})$ and $\widehat{F}_k(x_1,\ldots,x_n) = F_k(x_{h_1},\ldots,x_{h_n})$.

Therefore $\widehat{F}_i(x_1,\ldots,x_n) = \left(F_j(x_{h_1},\ldots,x_{h_n}) \wedge F_k(x_{h_1},\ldots,x_{h_n})\right)' = F_i(x_{h_1},\ldots,x_{h_n})$.

A gate node $(i, j, 0)$ in S at level n with one input j will have a global function $F_i = F_j'$ while the corresponding node in $\hat{S}$ will have a global function $\widehat{F}_i = \widehat{F}_j'$. The node j will have level less than n so by the induction assumption, $\widehat{F}_j(x_1,\ldots,x_n) = F_j(x_{h_1},\ldots,x_{h_n})$.

Therefore $\widehat{F}_i(x_1,\ldots,x_n) = \left(F_j(x_{h_1},\ldots,x_{h_n})\right)' = F_g(x_{h_1},\ldots,x_{h_n})$. □

Theorem 3.4. *For any function f which is symmetric in the set of variables* $\{x_i, x_{i+1},\ldots,x_j\}$ *if S is a network which implements the function f then any permutation of the set of input nodes representing the variables* $\{x_i, x_{i+1},\ldots,x_j\}$ *in S will also be an implementation of the function f.*

Proof: Let $(x_1,\ldots,x_n)$ be the set of variables on which the function f is defined. Let $(z_1,\ldots,z_n)$ be a permutation of the variables such that $x_i \rightarrow z_i$ and $f(x_1,\ldots,x_n) = f(z_1,\ldots,z_n)$ (i.e. only symmetric variables are permuted). Assume the input nodes in S are permuted in the same way to produce the network $\hat{S}$. By Lemma 3.5, the global functions at the output of S and $\hat{S}$ will be $f(x_1,\ldots,x_n)$ and $f(z_1,\ldots,z_n)$ which are equal. □

The networks S and $\hat{S}$ are called ***symmetric*** networks if $\hat{S}$ is simply the network S with some subset of the input nodes permuted and both S and $\hat{S}$ produce the same output function.

Once these symmetric networks are known to exist and the procedure is known for how one can be generated one from another, the search tree produced by BESS can be pruned to prevent all symmetric elementary networks from being generated. As long as one is generated all the others can be obtained by

making permutations. All symmetric partial networks are removed from the search to prevent all symmetric elementary networks from being generated. The description of how this pruning can be done is best explained with an example. A partial trace of the algorithm on the function $f = x_1' \vee x_2' \vee x_3'$ is given in Figure 3.23.

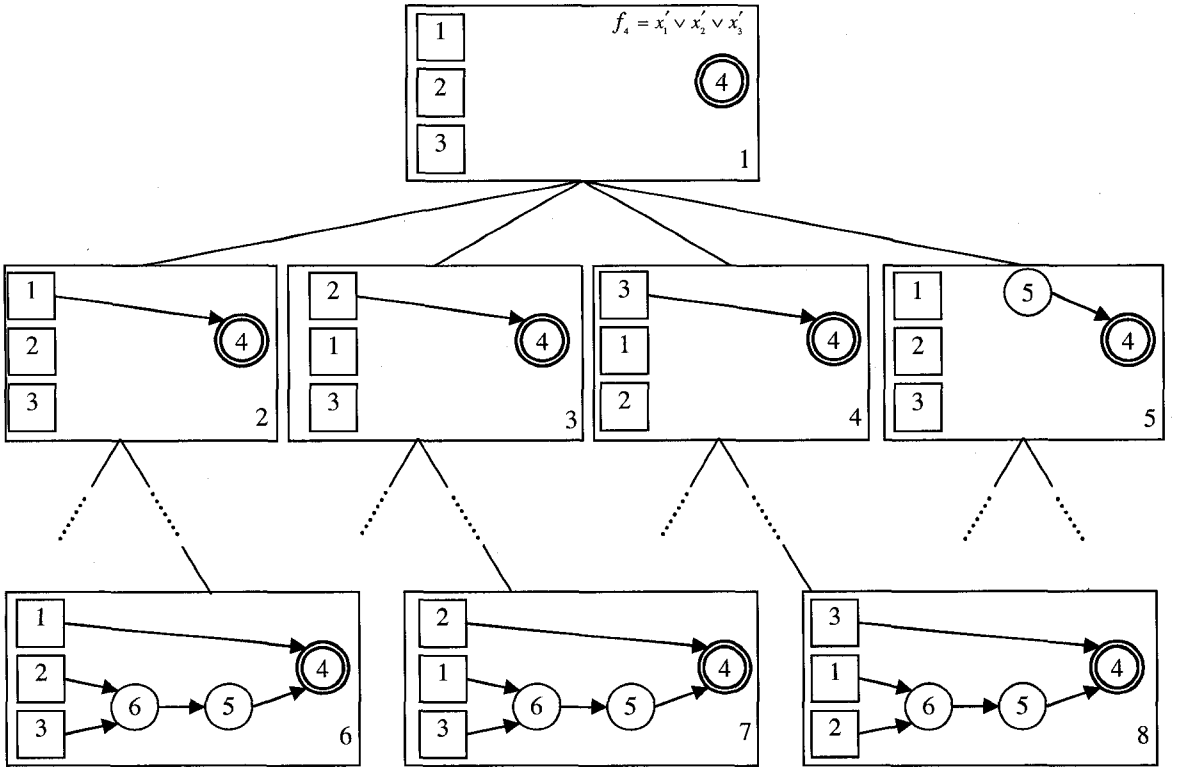


Figure 3.23: Trace of $f = x_1' \vee x_2' \vee x_3'$ for Output Symmetry Pruning

The algorithm is first called on network 1 in the top row. In this first call to the SynthesizeNetwork procedure, the minterm $x_1'x_2'x_3'$ from node 4 is chosen for covering. The possible covering options are found to be {1, 2, 3, new gate}. The output function $f = x_1' \vee x_2' \vee x_3'$ is symmetric in the variables x_1, x_2, and x_3. Theorem 3.4 implies that every elementary network contained in the set represented by the partial network 3 is symmetric to an elementary network represented by the partial network in 2 (and likewise for partial networks 4 and 2). Therefore there is no reason to continue to explore the solution space represented by the partial networks 3 and 4. These can be pruned immediately based on the symmetry of the output function.

In order for the algorithm to avoid symmetric partial networks of this type, a symmetric check in the FindConnectibleSet procedure can be added. This check should remove an input node from the connectible set if two conditions are satisfied:

(1) There must exist a second input node in the connectible set such that the output function of the circuit is symmetric in the primary inputs represented by these nodes.

(2) Both of these input nodes must have empty fan-out sets. Neither of these input nodes can be used in the partial network currently.

The second condition stipulates that this type of pruning can only be done the first time one of the symmetric input nodes is assigned as a fan-in to any node in the network. If the output function is symmetric with respect to the primary inputs x_1, x_2, and x_3 and the input node 1 has already been used as a fan-in of some node in the network, then a node that has the connectible set {1,2,3} must use nodes 1 and 2 for covering but can skip node 3. The symmetric pruning does not work for the pair of input nodes with one already used in the network. The elementary networks contained in the set represented by each of the partial networks will no longer be symmetric.

This idea of symmetric pruning can also be extended to networks with multiple output functions. In this case each of the output functions must be symmetric in the same set of variables in order for the pruning to work. The process remains the same for the common set of symmetric variables.

3.10 Summary

In this chapter we provided an in-depth description of the NAND2 synthesis algorithm, BESS. After an initial description of the core procedure, we addressed eight issues that arose from the simple algorithm in order to produce an optimal NAND2 network as efficiently as possible.

We first showed that the optimality of the network is guaranteed by using a branch-and-bound backtrack search method which searches the set of all possible network implementations. Next we discussed heuristic methods that can be used to aid in making the two decisions required during the execution of the algorithm. The choices made at these decisions effect the efficiency of the search. Implications of the decisions made by the algorithm were also discussed. Both structural and functional implications may result from a completed covering, and both can help to make decisions in the next set of covering steps. Conflicts may also result when a choice is made by the algorithm. We discussed how conflicts can occur and how BESS handles these situations.

In this chapter we also provided an analysis of BESS. First we analyzed the search space of the algorithm. This space is quite unique compared to other search algorithms in that it is constantly changing as synthesis proceeds. Two representations of the search space were given. Next, a proof of convergence was provided which guarantees that the search will terminate. A proof of completeness was provided to

guarantee that BESS will produce the optimal network with respect to the specified cost function (the number of gates in the network).

Finally, several pruning techniques were discussed. These techniques are able to prune significant portions of the search space by detecting and removing repeated networks. The use of these pruning techniques reduces the size of the search space and improves the efficiency of the algorithm.

Chapter 4

Optimal Synthesis Results with NAND2 Gates

In this chapter, results of the branch-and-bound exact synthesis algorithm, BESS, are presented. In Section 4.1 we will describe the various sets of functions that will be used to evaluate the algorithm. In Section 4.2 we will provide experimental justifications for the algorithmic improvements described in Chapter 3. In Section 4.3, results of BESS on various function classes will be presented. We will discuss these results both in terms of the complexity of the algorithm and in terms of the complexity of each function class. Finally, in Section 4.4 we will return to the search tree analysis from Chapter 3. Another estimate of the size of the search tree will be given based on the experimental data. Here, we will discuss the bounds on the run time and present experimental results to help demonstrate the complexity.

4.1 Experimental Data

The evaluation of BESS requires a variety of Boolean functions on which the algorithm can perform synthesis. In this section, we describe three categories of functions that will be used to execute this evaluation.

4.1.1 Representative Function Sets

An evaluation of the improvements added to BESS such as the decision heuristics, pruning techniques, and implications cannot be considered complete unless a wide variety of network structures and functional combinations are seen. One way to ensure this is to evaluate the algorithm on the set of all functions with n or fewer variables. This set will allow us to evaluate BESS on every possible function type. There are 2^{2^n} functions with n or fewer variables, however. This makes evaluating the algorithm on the entire set of functions with more than three variables difficult. Therefore we will select representative functions from the set of all n-input functions which will provide all possible structures found in the entire set of n-input Boolean functions but will be more manageable in size for the values of n we plan to consider. The Boolean function equivalence classes P and NPN [Harrison 63][Muroga 79][Slepian 53] are both good candidates for generating these representative functions.

Two Boolean functions are **P-equivalent** if one function can be transformed into the other by permuting the input variables. If two functions f and g are P-equivalent, then a NAND gate network can be created for g from the network producing f by performing the same permutation on the input variables as is required to

transform the function g to f. This implies that all functions contained in the same P-equivalence class will have the same NAND network structure. Therefore the set of n-input functions can be reduced down to only those which are not P-equivalent and the result will be a set of functions which will represent all structures possible for the entire set of n-input functions. Figure 4.4 compares the number of P-equivalence classes on n inputs to the number of Boolean functions dependent on all n variables for specific values of n. This table shows that the set of functions provided by the representatives from the P-equivalence classes greatly reduces the number of functions from the set of all n-input functions. However, the size of these sets is still be prohibitively large once $n = 5$. For this reason the NPN-equivalence is also considered.

Two Boolean functions are **NPN-equivalent** if one function can be transformed into the other by one or more of the following transformations: permutation of the input variables, negation of the input variables, negation of the output. Based on these transformations, if a NAND gate network is given for a Boolean function f then a NAND gate network can be created for any function NPN-equivalent to the function f by permuting the variables, and/or adding or removing NAND gate inverters at the inputs and/or output of the network.

The functions $f = x_1 \wedge x_2 \wedge x_3$ and $g = x_1 \vee x_2 \vee x_3$ are NPN-equivalent. The function g can be obtained from the function f by negating each of the variables and the output:

$$\begin{aligned} g &= x_1 \vee x_2 \vee x_3 \\ &= \left(x_1' \wedge x_2' \wedge x_3'\right)' \\ &= f'\left(x_1', x_2', x_3'\right) \end{aligned}$$

A NAND2 gate implementation of g can be created from a NAND2 gate implementation of f based on these same transformations. The network given in Figure 4.1 is a minimum NAND2 gate implementation of f.

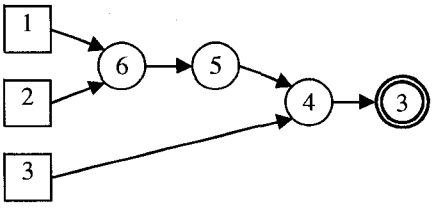


Figure 4.1: Minimum cost NAND2 network for $f = x_1 \wedge x_2 \wedge x_3$

A NAND2 gate implementation for g can be created from the network for f by removing the inverter (single input NAND2 gate) from the output and adding an inverter at each of the inputs. The resulting network is shown in Figure 4.2.

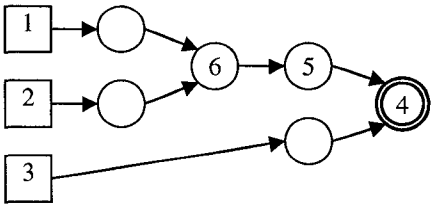
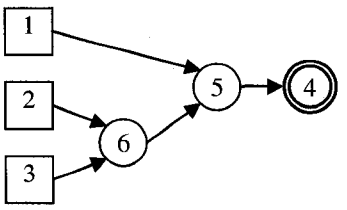
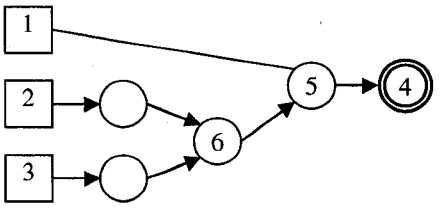


Figure 4.2: Minimum cost NAND2 network for $g = x_1 \vee x_2 \vee x_3$

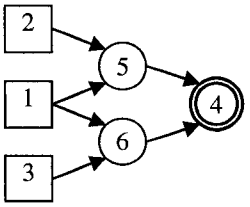
Unlike the P-equivalence class, functions within the same NPN-equivalence class may have different minimum networks with different structures and the structure obtained by transforming a NAND2 implementation for the one function in the class from a representative function may not always produce an optimal solution. For example, the functions $f = x_1x_2' \vee x_1x_3'$ and $g = x_1x_2 \vee x_1x_3$ are NPN-equivalent. However, the transformation of the minimal network for f to a network for g does not provide an optimal network for g. Figure 4.3 provides the optimal networks for functions f and g, networks (a) and (c) respectively. Network (b) is obtained by transforming the optimal network for f into a network for g. A comparison of networks (b) and (c) shows that the network which results from transformations within the NPN-equivalence class does not provide an optimal network for the NPN-equivalent function. A closer comparison of the networks (a) and (c) reveals that the basic structure inherent in the optimal networks for f and g are different.



(a) Optimal Network for $f = x_1x_2' \vee x_1x_3'$



(b) Network for $g = x_1x_2 \vee x_1x_3$ obtained by NPN transformations from $f = x_1x_2' \vee x_1x_3'$



(c) Optimal Network for $g = x_1x_2 \vee x_1x_3$

Figure 4.3: Networks for NPN-Equivalent Functions $f = x_1x_2' \vee x_1x_3'$ and $g = x_1x_2 \vee x_1x_3$

Some experiments on functions from the NPN-equivalence classes for n = 2, 3, and 4 show that out of 64,822 functions, 72% had a minimum network smaller than the network obtained by NPN transformations. Of these non-minimum networks, the largest difference between the derived network and the actual optimal implementation was 7 gates while on average the difference was 1.9 gates.

These experiments show that the NPN-equivalence classes cannot be used as a replacement for the entire set of functions as the P-equivalence classes can. Using the NPN-equivalence classes in this way would not allow all possible structures to be seen. However, the NPN-equivalence class can be used as a way to select a "good" representative subset for the entire set of functions. If a representative set for all n-input functions on which to run experiments is desired but the size of the set must be significantly smaller than what can be provided using the P-equivalence classes, then the NPN-equivalence classes are well suited. With this set, a good variety of the possible structures from the entire set of n-input functions will be seen but the size of the set will be drastically smaller than the set generated by the P-equivalence class.

The final column in Figure 4.4 gives the number of NPN-equivalence classes on the set of functions with n inputs for specific values of n. The size of the NPN sets created by selecting one representative function from each equivalence class for a given value of n will give a group of functions more manageable in size for performing experiments on larger values of n.

n	Number of Boolean functions with n or fewer variables	Number of Boolean functions which depend on exactly n variables	Number of P-equivalence classes on functions with exactly n variables	Number of NPN-equivalence classes on functions with exactly n variables
0	2	2	2	1
1	4	2	2	1
2	16	10	8	2
3	256	218	68	10
4	65,536	64,594	3,904	208
5	4,294,967,296	4,294,642,034	37,329,264	615,904
6	1.8×10^{19}	1.8×10^{19}	2.5×10^{16}	2.0×10^{14}

Figure 4.4: Size of Representative Sets

Two function sets have emerged from this discussion. The **P representative set** is composed of one function from each P-equivalence class for an input n. The **NPN representative set** is created similarly from the NPN-equivalence classes. The functions included in these representative sets are given in the Appendix.

4.1.2 Function Classes

The second group of functions used to evaluate BESS is composed of common Boolean functions. Each class will contain Boolean functions which evaluate a logic operation on an increasing number of inputs. The eleven function classes that will be used are: AND, NAND, OR, NOR, XOR, XNOR, MAJORITY, MULTIPLEXER, ADDER, DECODER, and THRESHOLD. The individual functions that make up each function class are given in the Appendix.

These function classes are used for two reasons. First the classes will be used to provide insight into the structures of minimum circuits. A structural analysis of these function classes will be performed based on the optimal networks provided by BESS. A general formula for the optimal cost of a function in a class can be extrapolated using these results. In some cases, this formula is proven to give the optimal cost as a

function of the number of inputs *n*. This analysis will allow us to then use the structure of these classes to relate and classify these and other classes of functions.

The results of BESS on these classes will also be used in the evaluation of the search performed by the algorithm. Since all of the functions in a given function class will have the same basic structure they can be used to estimate the size of the search space as a function of the number of inputs n.

4.1.3 Benchmark Functions

The final group of functions that will be used to evaluate BESS is a selection of industry benchmarks. The functions are selected from the set of MCNC benchmarks [Yang 91]. This group of functions will allow us to evaluate BESS on a larger variety of circuits. The goal will be to determine what benchmark functions the algorithm can complete and what structures these optimal networks will contain. There are a variety of functions contained in this suite so BESS will be evaluated on functions with both large and small numbers of variables, and also on multi-output functions.

4.1.4 Experiments

In the experiments that follow, the base algorithm will be the final version of BESS given in Chapter 3 including all pruning techniques, structural implications, and heuristics described there. Descriptions of alternate versions of the algorithm will be based on this version.

4.2 Specifics of the Algorithm

4.2.1 Decision heuristics

In Section 3.3 several heuristics for the two decisions that must be made during the course of the algorithm were described. This section provides experimental justification for the combination of heuristics which produce the smallest search tree. We use the representative sets outlined in Section 4.1.1 to compare the algorithm using different combinations of heuristic methods.

4.2.1.1 Minterm Selection Heuristics

In Section 3.3.1 we gave four options for a minterm selection heuristic. A description of why each of these methods would help to reduce the size of the search tree produced by the algorithm was also given. The heuristic options were:

SmallestCOV: From the set of uncovered minterms, select the minterm that has the *fewest covering options.*

DifficultCOV: From the set of uncovered minterms, select the minterm which has the most difficult *covering rank*: The four methods for performing a covering can be ranked according to their difficulty. From easiest to the most difficult:

1. Gate node that already exists as an input
2. Primary input node
3. Existing gate node
4. New gate node

SmallestFI: From the list of uncovered nodes, select the node that has the *smallest fan-in.* (A node may have 0,1,or 2 nodes in its fan-in). From the set of uncovered minterms for this node, choose an arbitrary minterm to cover.

SmallestCONN: From the list of uncovered nodes, select the node that has the *smallest connectible set.* From the set of uncovered minterms for this node, choose an arbitrary minterm to cover.

By combining these heuristics and adding the option of randomly selecting a minterm, the following minterm ordering schemes are created:

A. Randomly select a minterm for covering from all uncovered minterms (Random)
B. Select an uncovered minterm with the fewest covering options (SmallestCOV)
C. Select an uncovered minterm from a node with the smallest fan-in set (SmallestFI)
D. Select an uncovered minterm from a node with the smallest connectible set (SmallestCONN)
E. First select a node with the smallest fan-in set, (SmallestFI)
   then select an uncovered minterm from this node with the fewest covering options (SmallestCOV)
F. First select a node with the smallest connectible set, (SmallestCONN)
   then select an uncovered minterm from this node with the fewest covering options (SmallestCOV)
G. First find the set of nodes with the smallest fan-in, (SmallestFI)
   then from this set, select a node with the smallest connectible set (SmallestCONN)
   finally select an uncovered minterm from this node with the fewest covering options.(SmallestCOV)
H. Select an uncovered minterm with the most difficult covering label (DifficultCOV)
I. First find the set of minterms with the most difficult covering label (DifficultCOV)
   from this set, select the minterm with the fewest covering options (SmallestCOV)

The results of the experiments on these nine heuristic schemes are given in Figure 4.5. The evaluation is performed on the set of representative functions from the P-equivalence class for functions with 2, 3, and 4 inputs. There are a total of 3,980 representative functions from these 3 equivalence classes. Table (a)

provides the number of these functions on which the algorithm completed within the three hour time limit imposed for each instance. It also gives the total time required by the algorithm to attempt all 3,980 functions.

Since not all of the heuristics were able to complete the search for all functions, we took a subset of the 3,980 representative functions on which all heuristics were able to complete the optimal network search. The results presented in Table (b) provide the details of the search on these 933 functions. These details include the total amount of time it took for the algorithm to complete the search for all functions, the total number of nodes from all search trees produced by the algorithm on these functions, the average height of a path in these search trees, and the average width of a branch in these trees.

Table (c) provides details of the search for a subset of the heuristic methods. The three methods shown in this table completed all 3,980 functions from the P-equivalence classes. The details of the search on the entire set of P-equivalence classes is given for each heuristic method.

Heuristic	Networks Completed	Time for Completion
A	2,223	934.6 hrs
B	2,872	757.4 hrs
C	3,945	344.2 hrs
D	1,841	603.4 hrs
E	3,980	32.5 hrs
F	1,411	539.8 hrs
G	3,980	25.5 hrs
H	3,952	283.8 hrs
I	3,980	30 hrs

(a) Table of Results for all Minterm Heuristics

Heuristic	Networks Completed	Time for Completion	Size of Search Tree	Avg. Path Height	Avg. Branch Width
A	933	136 hrs	$9{,}443 \times 10^5$	18.478	2.575
B	933	16.4 hrs	$1{,}516 \times 10^5$	16.467	2.180
C	933	34.4 min	40×10^5	10.242	2.528
D	933	133.2 hrs	$9{,}608 \times 10^5$	18.945	2.243
E	933	5.1 min	8×10^5	9.381	2.314
F	933	223.8 hrs	$19{,}290 \times 10^5$	20.705	2.099
G	933	4.8 min	7×10^5	9.436	2.287
H	933	21.3 min	26×10^5	9.940	2.469
I	933	6.5 min	9×10^5	9.633	2.319

(b) Table of Results for all Minterm Heuristics on the 933 functions completed by all methods

Heuristic	Networks Completed	Time for Completion	Size of Search Tree	Avg. Path Height	Avg. Branch Width
E	3,980	32.5 hours	76,191,110	12.86	2.47
G	3,980	25.5 hours	63,097,950	12.95	2.41
I	3,980	30.0 hours	69,851,781	13.22	2.42

(c) Table of Results for Minterm Heuristics which completed all test functions

Figure 4.5: Minterm Heuristic Results

The results obtained from this experiment support the relationships discussed in Section 3.3.1 between the size of the search tree and the properties of the minterm heuristics. The first is a relationship between the number of covering possibilities for the chosen uncovered minterm and the width of a branch in the search tree. Based on this relationship, a heuristic method which uses the smallestCOV scheme should reduce the average branch width in the search tree and by extension reduce the size of the search tree. The experimental results show that this is the case. Heuristics B, E, F, and I have a smaller average branch width and a smaller total search tree size than their corresponding heuristics A, C, D, and H which do not use this minterm scheme.

The second relationship described in Section 3.3.1 exists between the height of the search tree and the possibility of covering minterms through functional implications. Heuristic methods smallestCONN and smallestFI were created to select nodes that have the least chance of being covered by functional implications. Here, the experimental results confirm that by using only smallestFI (heuristic C) compared to the random selection of a minterm (heuristic A) the average height of a path in the search tree is reduced and the total size of the search tree is also reduced. However, when smallestCONN is the only heuristic used, an increase in both the path height and search tree size results. The smallestCONN scheme is still a useful technique as long as it is used in conjunction with other methods. By combining the smallestCONN scheme with the smallestFI scheme (heuristic G) a further reduction of the search space can be obtained compared to when the smallestFI scheme is used alone (heuristic E).

The final relationship described exists between the type of coverings that can be performed and the size of the search tree. When the difficultCOV heuristic method is used to rank minterms based on the difficulty of covering the minterms, similar improvements on the search tree are gained as compared to those using the smallestFI methods over the random selection of a minterm.

Based on the results presented here, a combination of heuristic schemes provides the greatest reduction of the search tree. Both heuristic methods G and I use a scheme to minimize the width of a branch and the height of a path. While the heuristic for minimizing the height differs in these two heuristics, the same approximate improvement in the size of the search tree is achieved with each. Both G and I showed an average 100% improvement in the size of search tree over the random heuristic. Since G shows the better performance when all of the test functions are considered, we will use this as the minterm selection heuristic for the remainder of our experiments.

4.2.1.2 Network Ordering Heuristics

The second decision that must be made during the course of the search is the order in which the branches of the search tree are explored. Section 3.3.2 gave two heuristics which provided an ordering for how the partial networks created by a covering are explored.

CovOrder: Order the nodes available for covering according to their *covering rank*: (1) a primary input node, (2) a gate node that already exists as an input, (3) an existing gate node, (4) a new gate node. Nodes with lower covering rank are chosen first.

CostOrder: Order the nodes available for covering according to the cost of the partial networks that result when the node is used for covering. Networks with lower cost are chosen first

Based on these two heuristic methods, the following ordering heuristics were created:

A. Random (Random)
B. Order only the connectible set (CovOrder)
C. Order the networks based only on the cost (CostOrder)
D. Order both the connectible set and then the networks based on their cost (CovOrder), (CostOrder)

The results of the experiments with these network ordering schemes are given in Figure 4.6. The evaluation is performed on the set of representative functions from the P-equivalence class for functions with 2, 3, and 4 inputs. The first column in the table indicates the heuristic used during the search. The following five columns give the details of the search. These details include the total amount of time it took for the algorithm to complete the search, the total number of nodes from all search trees produced by the algorithm, the average height of a path in these search trees, the percentage of the search tree completed before the optimal network was found, and a measure of the distance the initial network found by the algorithm was to the optimal network.

Heuristic	Time for Completion	Size of Search Tree	Average Path Height	Portion of Search Tree Completed to Optimal Network	Avg. Difference of Initial to Optimal Cost
A	36.1 hours	245×10^6	12.244	45%	4.384
B	32.6 hours	215×10^6	12.169	47%	4.366
C	28.4 hours	205×10^6	12.987	34%	4.912
D	25.7 hours	169×10^6	12.953	32%	4.952

Figure 4.6: Results of Network Ordering Heuristics

These results show that the heuristic scheme D performed the best, however not for the reason proposed in Section 3.3.2. The ultimate goal of the heuristic methods is to shrink the size of the search tree which does indeed happen with heuristic D (by 31% compared to random). However, the last three columns of the table shows that this reduction is a result of the optimal network being found sooner in the search rather than as a result of a smaller cost for the initial complete network or a reduction in the height of the paths in the search tree as hypothesized in Section 3.3.2.

A comparison of Figure 4.5 and Figure 4.6 shows that the network ordering does not have as large of an impact on the size of the search tree as the selection of the minterm for covering does. On average, the reduction in the search tree size for the best minterm heuristic version over random selection was 1,134,501 while the reduction in the search tree size for the best network heuristic versions over random selection was

only 6,693. This difference is a result of the fact that all of the choices for covering must be explored no matter what order is chosen, while only one of the many possible minterms is selected for covering. The network order heuristic can only effect where in the search the optimal network will be found, while the minterm selection can effect the height of every path and the width of every branch in the search tree.

4.2.2 Structural Implications

In this section, we will provide experimental evidence to support the use of the structural implications described in Section 3.4.3 to reduce the size of the search space. Structural implications provide additional information about a partial network which can then be used to reduce the height of the search tree. The idea behind these implications is that new gate nodes can be added to a partial network whenever they become necessary. This will eliminate the nodes in the search tree with a branch width of one. In addition these implications also increase the possibility that a partial network can be pruned earlier further reducing the size of the search tree.

The results of the experiments on structural implications are given in Figure 4.7. In these results, the nodes of the search tree are divided into three types. A branch node is a node of the search tree where there exists at least two ways of covering the selected minterm. An implication node is a node of the search tree where only one possibility for covering exists. A leaf node is a node of the search tree which is either a solution or where the algorithm can prune based on the cost of the network. In this experiment a version of the algorithm which uses structural implications is compared against a similar version without structural implications. Once again the evaluation is performed on the set of representative functions from the P-equivalence classes for 2-, 3-, and 4-input functions.

Structural Implications?	Networks Completed	Time for Completion	Size of Search Tree			Structural Implications	Avg. Path Height
			Branch	Implication	Leaf		
Yes	3,980	21.0 hrs	59,528,859	77,365,086	96,952,916	73,771,080	12.94
No	3,970	99.6 hours	233,014,252	203,973,773	374,975,848	–	18.60

Figure 4.7: Experimental Results of Structural Implications

The number of branches in the search tree is reduced 74% through an increase in the ratio of implications to branches performed during the search. By performing implications as soon as they become available, some partial networks can be pruned earlier based on cost, thus reducing the overall size of the search tree. Since the algorithm was able to reduce the size of the search tree with little or no effect to the time needed to complete the search, structural implications are an effective addition to BESS.

4.2.3 Pruning

Three pruning techniques were presented in Section 3.9 which remove repetitive portions of the search tree. The first technique was based on overlapped covering. This type of repetition is removed by adding the minterm m to the on-set of the existing fan-in node l for the remaining coverings of m after l. The second pruning technique was based on the symmetry of the NAND gate. This type of repetition is removed

through the use of the non-connectible set. Experimental results evaluating the use of these two pruning techniques to improve the search tree size are summarized in Figure 4.8. The first two columns of the table indicate the combination of pruning techniques used by the algorithm. The third column gives the number of functions from the P representative set that the algorithm was able to complete within the time limit set for each instance. The last four columns of the table present the results of the algorithm on only those functions (3,976) which all versions were able to complete.

Overlapped Covering	Gate Symmetry	Networks Completed	Time for Completion	Size of Search Tree	Overlapped Cover Prevention	Symmetric Prevention
No	No	3,976	55.8 hours	391,368,196	-	-
No	Yes	3,980	41.6 hours	312,657,042	-	81,271,926
Yes	No	3,980	27.6 hours	195,711,549	5,422,268	-
Yes	Yes	3,980	20.8 hours	158,742,445	2,868,728	46,371,311

Figure 4.8: Experimental Results of Pruning Techniques

These results confirm what was proposed in Section 3.9, the search tree is reduced by using both pruning techniques. In addition, any time added to each step of the algorithm in order to perform the pruning is more than made up for in the amount of time saved by shrinking the search tree.

When the overlapped covering technique is used by itself, a 50% reduction is made in the size of the search tree. While it is difficult to count the number of times pruning is actually performed a count of the number of times the steps are performed to prevent this type of repetition can be made. For an average function, overlapped covering prevention was performed in 6.94% of all calls to the SynthesizeNetwork procedure.

When only the pruning technique based on the symmetry of the NAND gate is used, an improvement is again made in the size of the search tree: a 20% reduction in the total size of the search tree. The reduction is smaller than that made by the overlapped covering technique but is still significant enough to be used as a way to improve BESS. The steps needed to prevent this type of repetition were performed in 67% of the calls to the SynthesizeNetwork procedure. Thus, while the steps to prevent this type of repetition were performed quite often a smaller savings was seen compared to the previous method.

The final row of the table in Figure 4.8 gives the results of the algorithm when both pruning techniques are used. The data in this row show that further gains can be made in reducing the size of the search tree by combining these two techniques. The size of the search tree is reduced by 59% compared to the search tree when no pruning techniques are used, 49% when only overlapped covering pruning is used, and 19% when only gate symmetry pruning is used.

The third pruning technique described Section 3.9 is based on the symmetry of the output function. This pruning technique removes repetition in the search tree by forcing the algorithm to only generate one network from the set of symmetric networks. Since this pruning technique can only be performed when symmetries exist in the output function, this pruning technique was only tested on functions that contain

some input symmetry. Figure 4.9 shows the number and type of symmetric functions that exist in each of the representative sets.

Inputs	Size of Representative Set	Functions with no Symmetry	Max Symmetry = 2	Max Symmetry = 3	Max Symmetry = 4
2	8	2	6	-	-
3	68	14	40	14	-
4	3904	2209	1509	172	14

Figure 4.9: Symmetric Functions in Representative Sets

The experimental results of the algorithm both with and without this symmetric pruning technique are given in Figure 4.10. The last three columns of the table present the results of the algorithm on only those functions (1,754) which both versions were able to complete. Once again, a decrease in the size of the search tree is achieved when the pruning is added to the algorithm. On average a 10% reduction in the size of the search tree occurs for an individual function. Counting the number of times input variables are left out of the connectible set (column 5) gives an indication of how often this type of pruning is performed. In the functions where symmetric variables exist, this type of pruning is performed in only 0.3% of all calls to the SynthesizeNetwork procedure.

Symmetry	Networks Completed	Time for Completion	Size of Search Tree	Symmetric Prevention
No	1754	22.5 hours	170,421,562	0
Yes	1755	15.8 hours	118,186,427	137,106

Figure 4.10: Pruning Based on Symmetry of the Output Function

Figure 4.11 presents the results of the algorithm using each possible combination of pruning techniques. These results show that each pruning technique helps to reduce the size of the search tree. Therefore to complete the search in the shortest amount of time all pruning techniques should be used. The last five columns of the table present the results of the algorithm on only those functions (3,973) which all versions were able to complete.

Overlapped Covering	Gate Symmetry	Function Symmetry	Networks Completed	Time for Completion	Search Tree Size	Overlapped Cover Prevention	Gate Symmetry Prevention	Function Symmetry Prevention
No	No	No	3,974	70.4 hrs	484,519,235	0	0	0
No	No	Yes	3,976	51.5 hrs	362,914,729	0	0	111,238
No	Yes	No	3,975	45.5 hrs	348,396,748	0	88,526,077	0
No	Yes	Yes	3,980	39.0 hrs	294,877,292	0	76,051,744	263,127
Yes	No	No	3,976	30.9 hrs	218,295,023	6,091,604	0	0
Yes	No	Yes	3,980	26.0 hrs	185,138,381	5,132,389	0	81,545
Yes	Yes	No	3,979	22.9 hrs	178,037,335	3,098,995	50,948,072	0
Yes	Yes	Yes	3,980	19.6 hrs	150,908,100	2,752,075	43,907,273	153,940

Figure 4.11: Results of All Pruning Types on the Size of the Search Tree

4.2.4 Global Functional Implications

In Section 3.4.2 two types of global functional implications that could be added to the algorithm to help reduce the size of the search space were described. These functional implications were based on a

reconvergent structure that exists in the partial network. These functional implications provide additional information about the Boolean functions for some gate nodes in the partial network. This helps to reduce the size of the search space by limiting some of the covering options in the partial network.

The results of the experiment using the two types of global functional implications are given in Figure 4.12. In this experiment a version of the algorithm which uses each type of functional implication is compared against a version using both types and a version using neither. Once again the evaluation is performed on the set of representative functions from the P-equivalence classes for 2-, 3-, and 4-input functions.

Simple Pattern	General Pattern	Networks Completed	Time for Completion	Size of Search Tree	Implications based on Simple	Implications based on Extended
No	No	3,980	27.4 hours	207,166,060	-	-
No	Yes	3,980	37.5 hours	167,212,434	-	35,649,311
Yes	No	3,980	26.0 hours	186,786,193	8,657,042	-
Yes	Yes	3,980	39.5 hours	172,282,510	9,131,129	27,284,213

Figure 4.12: Experimental Results for Functional Implications from Bridges

These results show that the each type of functional implication is able to reduce the size of the search tree. Using general pattern alone the search space is reduced by 19%, while a 10% reduction occurs when only the simple pattern is used. A larger reduction occurs when the general pattern is used since this method encompasses more structures. When functional implications based on this general pattern are added to the algorithm, however, the time for the algorithm to complete is increased. The algorithm requires an average of 8.1 seconds to produce 10,000 search tree nodes when using general pattern for global functional implications compared to requiring an average of only 4.8 seconds for every 10,000 search tree nodes when no global implications are performed. The increase of time occurs because this type of implication is difficult to detect, so more work must be performed each time a global functional implication is sought. Only a slight increase in the time per search tree node is found when implications based on the simple pattern are made. A simple method is used for finding these structures and a fast method is used to detect and perform the implications.

In this implementation of the general pattern for global functional implications, a check for every node in the network is performed to determine what structures exist in the network. However, the only possible structural change made in the network once a single covering is made is the addition of a single edge. Therefore if new global functional implications were to exist they should be based on a reconvergent fan-out structure that would result from this new covering. Thus a second implementation of the general reconvergent pattern was created which only searches for the reconvegent pattern in a set of nodes which contains the node l used to cover the minterm. The same experiments as in Figure 4.12 were performed using this new implementation. The results are given in Figure 4.13. These results show an improvement in the time for completion (about a 12% reduction) than what was seen in the previous results. A decrease in the number of implications performed (by more than 25%) also occurs. While the time per search tree

node was reduced to 7.2 seconds for 10,000 search tree nodes, this modification still does not provide enough of a reduction in the size of the search space to make up for the increased time for performing the implications. Therefore global functional implications based only on the simple reconvergent pattern are a worthwhile improvement to BESS.

Simple Bridge	Extended Bridge	Networks Completed	Time for Completion	Size of Search Tree	Implications based on Simple	Implications based on Extended
No	No	3,980	27.4 hours	207,166,060	-	-
No	Yes	3,980	33.3 hours	166,485,877	-	26,752,015
Yes	No	3,980	26.0 hours	186,786,193	8,657,042	-
Yes	Yes	3,980	34.6 hours	169,740,706	8,990,562	19,791,640

Figure 4.13: Functional Implication Experiments with Modified Extended Bridge

4.3 Optimal Results

4.3.1 Optimal Results for Equivalence Classes

One of the goals for exact synthesis is to provide a database of minimum circuits. Several works previous to this have completed such databases with different constraints.

Ninomiya [Ninomiya 61] provides a catalog of minimal {NOR, NAND} networks for all functions with 4 or fewer variables. The cost function used to optimize the networks is based on the number of transistors required to complete the circuit. Thus the cost of an n-input NAND or NOR gate is n, the number of inputs to the gate. In addition to the NAND and NOR gates, Ninomiya allows the use of a wired-AND at no cost. Under these constraints, he completes the catalog for all 222 NPN-representative functions of 4 or fewer inputs.

Hellerman [Hellerman 63] uses an explicit enumeration method to give a catalog of minimal NOR network and minimal NAND networks for all functions with 3 or fewer variables. The cost criteria placed on these networks prioritized the number of gates in the network first and the number of connections in the network second. In these networks, both a fan-in and fan-out restriction of 3 is placed on the networks. He completes the catalog for 79 out of the 80 P-representative functions of 3 inputs or fewer. The three-input XOR function required more than 7 gates with the restrictions placed on the network and was not completed in the time allotted.

Smith [Smith 65] uses the same method to give a similar catalog of minimal NOR networks and minimal NAND networks for all functions with 3 or fewer variables when both complemented and uncomplemented inputs are available and no restriction is placed on the fan-in and fan-out of the gates. In this catalog, networks for all 80 P-representative functions are given with the largest network requiring 5 gates.

Baugh et. al. [Baugh 69] use an implicit enumeration method to provide a catalog of minimal {NOR, AND} networks for all functions with 3 or fewer variables. For these networks, no fan-in or fan-out restrictions are applied. The cost function is based first on the number of gates in the network and second

on the number of connections. Optimal networks for all 80 representative functions from the P-equivalence classes were found. The maximum number of gates in a network is 6. A second catalog of results was presented using this method as well. [Culliney 71] gives a catalog of 438 minimal {NOR} networks for functions with 4 or fewer variables towards the goal of completing all 3,984 functions.

Davidson [Davidson 68b] uses a branch-and-bound method similar to the one described here to give a catalog of minimal {NAND} networks for all functions with 3 or fewer variables. In this work, optimality is determined only by the number of gates in the network. No limits on the fan-in set of the gate nodes or the levels of the network are imposed. Optimal networks for all 77 functions were found. The maximum cost for an optimal network was seven gates.

Culliney et. al. [Culliney 79] also uses a similar branch-and-bound method to provide a catalog of minimal networks. These networks are built from AND and OR gates. The catalog presented contains networks for all functions with four or fewer variables. Inputs to the networks are available in both complemented and uncomplemented form. Optimality is determined first by the number of gates, and second by the number of connections. No fan-in or fan-out restrictions are imposed on the gates. Since both complemented and uncomplemented variables are available only representative functions from the NPN-equivalence classes were completed. Optimal networks for all 222 functions are found with the exception of the function $x_1 \oplus x_2 \oplus x_3 \oplus x_4$ where the optimal network with respect to the number of gates only is found. The maximum number of gates in a network here is 9.

We present yet another catalog of optimal networks extending this previous work. The catalog presented here will use NAND2 gates and a cost function which depends only on the number of gates in the networks. The variations of BESS required to match the synthesis options used in these previous works will be presented in Chapter 5. Due to space constraints, the complete databases under these constraints are not provided here, but are given at www.eecs.umich.edu/~ebroerin/OptimalSynthesis.

As described in Section 4.1.1, optimal results for only one representative function from each P-equivalence class needs to be obtained in order for optimal networks of all functions on n inputs to be completed. Using these representative sets and BESS a table of results for all n-input functions can be created. Such a table for 2-input functions is given in Figure 4.14. The far left column gives the binary representation for the Boolean function while the second column gives the integer representation taken from this binary representation. These representations are obtained from the truth table for the function with the most significant digit representing the truth table line $x_1'x_2'$ and the least significant digit representing the truth table line x_1x_2. For example the function $x_1 \vee x_2$ has the truth table:

x_1	0	0	1	1
x_2	0	1	0	1
$x_1 \vee x_2$	0	1	1	1

Therefore the binary representation of $x_1 \vee x_2$ is 0111 and the integer representation is 7.

Since optimal networks for only a subset of the two-input functions are given, the remaining functions are indicated according to their P-equivalence class representative. These are given in the third column of the table. The P-equivalent relationship implies that optimal networks for the functions indicated in this column can be obtained from the optimal network given by permuting the input variables. The final two columns of the table give the optimal network for the function both in a text and graphical format. In some cases, more than one network structure has optimal cost for a function. Only one of these will be presented here, however the number of such networks is given in column 4.

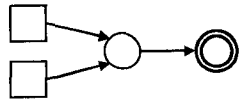
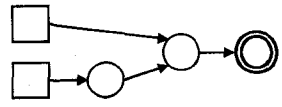
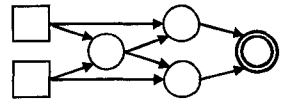
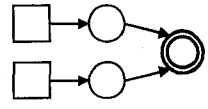
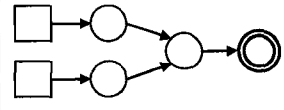
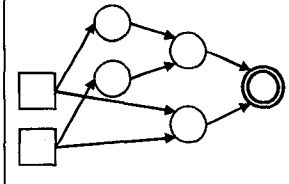
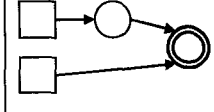
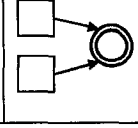
Function		P-Equivalent Functions	Optimal Network		
Binary	Integer		Number	Text Format	Graphical Format
0001	1		1	O = NAND(I1) I1 = NAND(a, b)	
0010	2	4	1	O = NAND(I1) I1 = NAND(a, I2) I2 = NAND(b)	
0110	6		1	O = NAND(I1, I2) I1 = NAND(a, I3) I2 = NAND(b, I3) I3 = NAND(a, b)	
0111	7		1	O = NAND(I1, I2) I1 = NAND(a) I2 = NAND(b)	
1000	8		1	O = NAND(I1) I1 = NAND(I2, I3) I2 = NAND(a) I3 = NAND(b)	
1001	9		3	O = NAND(I1, I2) I1 = NAND(a, b) I2 = NAND(I3, I4) I3 = NAND(a) I4 = NAND(b)	
1011	11	13	1	O = NAND(I1, b) I1 = NAND(a)	
1110	14		1	O = NAND(a, b)	

Figure 4.14: Optimal Networks for all 2-Input Functions

Similar tables for 3-, 4-, and 5-input functions are given in the Appendix. Optimal networks for all 3-input functions are given but due to space considerations, only the optimal cost for the 4- and 5-input functions are given. Networks for all of these functions can be found at www.eecs.umich.edu/~ebroerin/OptimalSynthesis. Both the number and size of the 5-input representative

functions prohibit us from completing all these optimal networks. However, BESS was run on 5,533 functions from this class. We limited the amount of time that the algorithm was allowed to search for an optimal network to 48 hours. Out of 5,533 five-input functions, BESS completed the search on 4,745 of them. The table given in the appendix includes the functions on which BESS did not complete the search. A star has been placed next to them to indicate they were not completed. For these functions, the network cost shown in the table is the cost of the smallest network found during the search before it was terminated.

The graphs shown in Figure 4.15 represent all the data presented in the appendix. The cost of the optimal network for each function completed is shown. These graphs give an indication of the size of an average network for each of the input classes. The maximum cost of a 3-input network was 10 while the minimum was 2 and the average was 5.96. For four-input functions, the maximum cost network was 14, the minimum was 3, and the average was 9.72. Of the five-input networks found, the maximum cost was 16, the minimum was 5, and the average was 11.5.

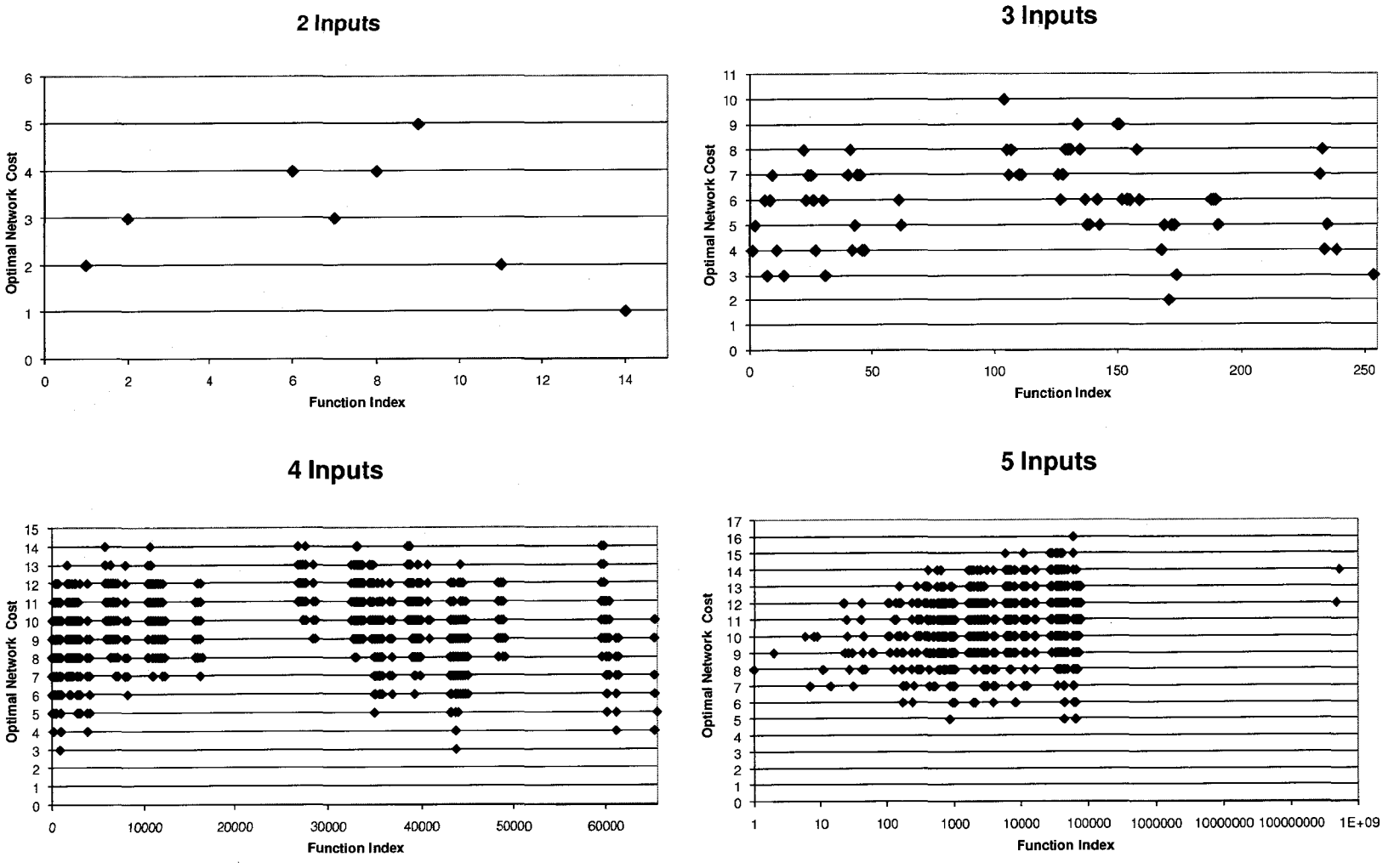


Figure 4.15: Graph of Optimal Network Cost for Equivalence Class Functions

The size of the networks completed gives an indication of the improvements that have been made over previous work. Earlier works were only able to complete networks with a maximum of 9 gates. A combination of improved computing power as well as improvements made to the algorithm has allowed us to extend the database of optimal circuits.

4.3.2 Optimal Results for Function Classes

Results of BESS on the function classes described in Section 4.1.2 can be used in several ways. First, they give an indication of the complexity of the algorithm and can show how large a function the algorithm can complete. In addition, they can show what factors play a role in the size of the search tree produced by the algorithm across all the classes. In this section, we will use the results of BESS on the function classes to gain insight into the structure of minimum circuits. Using the optimal networks found by BESS, we can gain an understanding of the complexity of the circuits for each class by finding the patterns in the structures that exist in these networks as well as a formula to determine the cost of the network.

For each function class, we will give a table of results containing the cost of the optimal network and details about the algorithm's search for functions from this class. Circuit diagrams for the smaller functions from the class will also be given. Some function may have multiple networks with minimum cost. In this case each of the networks will be provided in the diagram. These diagrams will help to illustrate the structural pattern that emerges for many of the classes. Based on the results and diagrams a formula for the cost of the optimal network will be presented for a function in the class with n inputs. After the results for each function class have been presented, we will conclude with a discussion of these results and what they mean in terms of the complexity of the algorithm as well as the complexity of the function classes.

4.3.2.1 AND Class

The table in Figure 4.16 gives the results of BESS on the functions from the AND class. The algorithm was able to complete the search for functions up to 7 inputs with the largest network containing 12 gates. The search for an optimal network for the 8-input function was stopped after three days with the smallest network found thus far containing 14 gates.

Inputs	Cost of Optimal	Time for Completion	Search Tree Size
2	2	0 s	5
3	4	0 s	7
4	6	0 s	25
5	8	0 s	310
6	10	9 s	13,885
7	12	16.2 min	1,401,599
8*	14	3 days	284,712,649

Figure 4.16: AND Class Results

Figure 4.17 gives the optimal networks for the AND functions when $n = 2,3,4$. Two minimum-cost networks are given for the 4-input function. A pattern emerges within the structure of these optimal

networks. Each network is a combination of $(n - 1)$ two-input AND networks. Since a two-input AND network requires two NAND2 gates the cost of an optimal network of an n-input AND function is $2(n - 1)$.

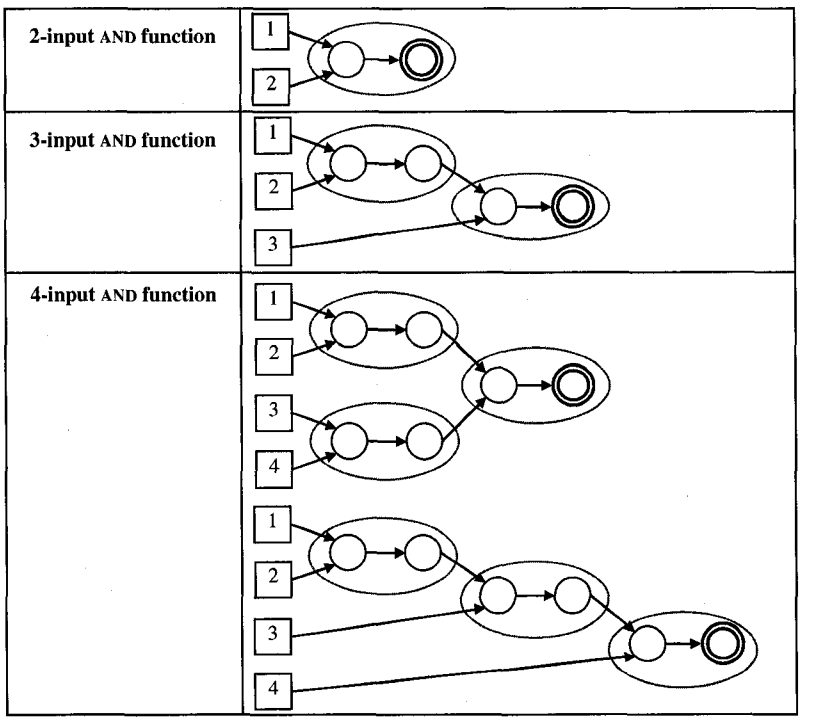


Figure 4.17: Optimal Networks for AND Class

This construction gives an upper bound on the optimal cost of an n-input AND network, $|\text{AND}_n|$. The following shows that this formula also gives a lower bound on the optimal cost.

Theorem 4.1. *The number of gates required in an optimal network for an n-input AND function is at least* $2(n-1)$. $|\text{AND}_n| \geq 2(n-1)$.

Proof: (by induction) The results of the algorithm prove that the optimal network for a 2-input AND function is the network shown in Figure 4.17 and requires 2 gates. Thus $|\text{AND}_2| \geq 2$.

Assume for some integer $m > 2$, $|\text{AND}_m| \geq 2(m-1)$.

Let $n = m + 1$ and let C be a minimal size network for AND_n. Since C is a minimal size network for AND_n, the function realized by C is $f_C = x_1 \wedge \ldots \wedge x_n$.

We will first establish some properties of the network. Assume there exists a gate $(i, j, 0)$ in C that contains only a single input node in its fan-in set. Let the primary input represented by this input node be x_j. i cannot be the output of the network since $n > 2$ therefore there must be a gate node (l, i, k) in C that contains i in its fan-in set. This portion of the network C is shown in Figure 4.18.

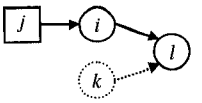


Figure 4.18: Part of the Network C

Consider the 2 cases when x_j is replaced by a constant. If x_j is replaced by the constant 1 then $f_i = 0$ and $f_l = 1$, no matter what else is connected to the gate l. Therefore, the edge (k, l) is unnecessary in this case. In the second case, if x_j is replaced by the constant 0 then $f_i = 1$ which implies $f_l = f_k'$. When x_j is replaced by the constant 0, however, the network will realize the function $f_C = 0$. In this case, this entire portion of the network could be replaced by the constant 0. Therefore the edge (k, l) is unnecessary in this case as well. These two cases imply that an edge (k, l) is redundant. This edge can be removed from the network while still maintaining the functionality of the circuit. Therefore we can assume when the node $(i, j, 0)$ exists in the network, then the node $(l, i, 0)$ must also exist. This situation however results in a double negation in the network: $f_i = x_j'$ and $f_l = f_i' = x_j$. Therefore both gates i and l can be removed from the network C with j replacing l. The result is a network that implements AND_n and contains 2 fewer gates than C. This would imply that C is not an optimal cost network for AND_n. Therefore the assumption that such a gate $(i, j, 0)$ exists in the network is false.

According to the previous argument, there must exist a gate (i, j, k) in the network such that j and k are primary inputs. Through the symmetry of the AND_n function we can assume that $j = 1$ and $k = 2$. Since $n > 2$, i cannot be the output to the network. Thus, there must exist some gate (l, i, h). In this case, $f_i = x_1' \vee x_2'$ and $f_l = f_i' \vee f_h'$. If either x_1 or x_2 is replaced by the constant 0 then $f_i = 1$ and $f_l = f_h'$. Since the network will realize the function $f_C = 0$ in these cases, the edge (h, l) is not necessary. If both x_1 and x_2 are replaced by the constant 1 then $f_i = 0$ and $f_l = 1$. Again the edge (h, l) is unnecessary in this case as well. Therefore, the edge (h, l) is redundant. This edge can be removed from the network while still maintaining the functionality of the network. Hence we can conclude that the structure of this part of the network must contain the nodes $(i, 1, 2)$ and $(l, i, 0)$. This structure is shown in Figure 4.19. l must be the only fan-out of i since the above argument will hold for any gate which contains i in its fan-in set.

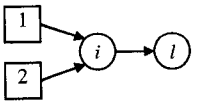


Figure 4.19: Part of the Network C

Now that this structure of C has been established, consider the network $\tilde{C}$ created from C where x_1 is replaced by the constant 1. $f_i = x_2'$, $f_l = x_2$, and the network will realize the function $f_{\tilde{C}} = x_2 \wedge \ldots \wedge x_n$. In this new network, the gates i and l are redundant and can be removed with l replaced by the node 2. The result will be a network that implements the function AND_{n-1} and contains

2 less gates than C. By the induction assumption, $|AND_{n-1}| \geq 2(n-2)$. So $|\tilde{C}| \geq 2(n-2)$. Therefore $|AND_n| = |C| \geq 2(n-2)+2 = 2(n-1)$. □

4.3.2.2 OR Class

The table in Figure 4.20 gives the results of BESS on the functions from the OR class. The algorithm was able to complete the search for functions up to 6 inputs with the largest network containing 15 gates. The search for an optimal network for the 7-input function was stopped after three days with the smallest network found thus far containing 18 gates.

Inputs	Cost of Optimal	Time for Completion	Search Tree Size
2	3	0 s	5
3	6	0 s	16
4	9	0 s	129
5	12	4 s	6,796
6	15	29.7 min	2,527,493
7*	18	3 days	312,663,276

Figure 4.20: OR Class Results

Figure 4.21 gives the optimal networks for the OR function when $n = 2,3,4$. These diagrams show that the optimal network for an n-input OR function is a combination of $(n-1)$ two-input OR networks. Since a two-input OR network requires three NAND2 gates, the cost of the optimal network of an n-input OR function will be $3(n-1)$.

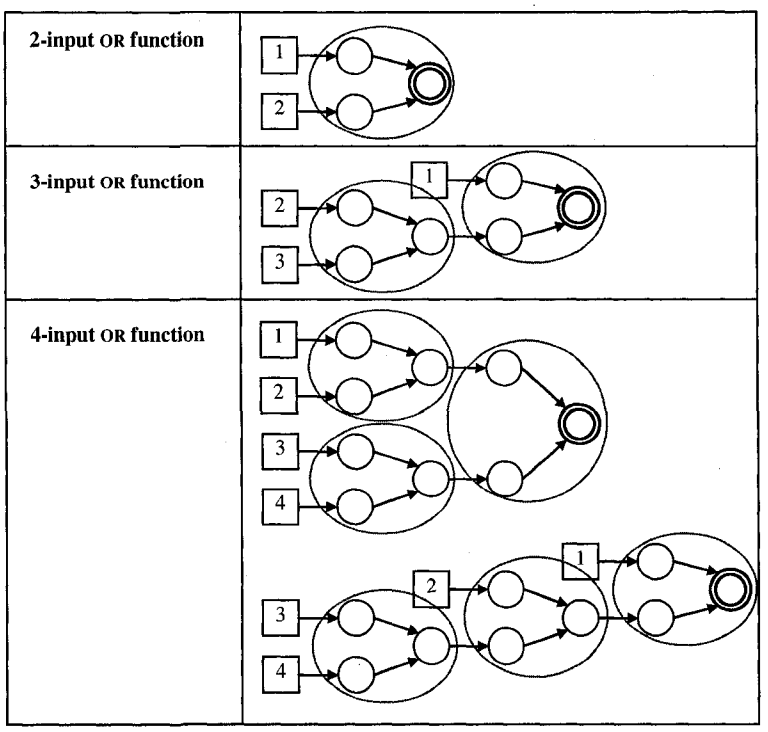


Figure 4.21: Optimal Networks for OR Class

Once again, this construction gives an upper bound on the optimal cost of an n-input OR network, $|\mathrm{OR}_n|$. The following shows that this formula also gives a lower bound on the optimal cost.

Theorem 4.2. *The number of gates required in an optimal network for an n-input OR function is at least 3(n-1).* $|OR_n| \geq 3(n-1)$.

Proof: (by induction) The results of the algorithm prove that the optimal network for a 2-input OR function is the network shown in Figure 4.21 and requires 3 gates. Thus $|\mathrm{OR}_2| \geq 3$.

Let $n > 2$, and assume that for $2 \leq m < n$, $|\mathrm{OR}_m| \geq 3(m-1)$.

Let C be a minimal size circuit for OR_n. The function realized by C is $f_C = x_1 \vee \ldots \vee x_n$.

Let us first establish some properties of the network C. There must exist a gate $(i, j, 0)$ in C that contains only primary inputs as fan-in. Assume (i, j, k) has two inputs x_j and x_k. If x_j is replaced by the constant 1 then $f_i = x_k'$ and C will realize the function $f_C = 1$. Therefore the second fan-in to i is not needed in this case. If x_j is replaced by the constant 0, $f_i = 1$ and the network C will realize the function $f_C = x_1 \vee \ldots \vee x_{j-1} \vee x_{j+1} \vee \ldots \vee x_n$. Therefore the input node x_k is used elsewhere in the network and is not needed as a fan-in to node i in this situation either. These two cases imply that the edge (k, i) is redundant. This edge can be removed from the network while still maintaining the functionality of the circuit. Therefore we can assume that the only fan-in to the gate node i is j. This argument proves that

any gate in C which contains a primary input in its fan-in set contains only this input node in its fan-in set.

Let the $(i, 1, 0)$ be the gate from C which contains the primary input x_1 in its fan-in set. The gate i can not be the output of the network since $n > 2$. Let gate (j, i, k) be a gate in the network that contains i as a fan-in. j must contain a second node, k, in its fan-in set and by the previous argument it must be a gate node. Based on the symmetry of the OR function, we can assume that $(k, 2, 0)$ also contains an input node in its fan-in set and that j is not the output node for the network either since $n > 2$. Thus there must be a gate node (l, j, h) from C that contains j as a fan-in. This portion of the network is shown in Figure 4.22.

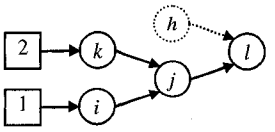


Figure 4.22: Part of Network C

Now we can show that the edge (h, l) is redundant. If the input nodes 1 and 2 are replaced by the constants 0 and 1, the gates i, j, k, and l and the network C will compute the following values:

x_1	x_2	f_i	f_k	f_j	f_l	f_C
0	0	1	1	0	1	$x_2 \vee \ldots \vee x_n$
0	1	1	0	1	f_h'	1
1	0	0	1	1	f_h'	1
1	1	0	0	1	f_h'	1

In each case, the edge (h, l) can be removed from the network and f_C will remain unchanged. Therefore we can assume that this portion of the network will have the structure shown in Figure 4.23. This property will be true for any gate in the fan-out of j, so l must be the only fan-out of j.

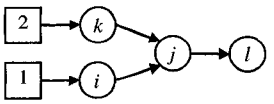


Figure 4.23: Part of Network C

Let $\tilde{C}$ be the network C with input node 1 replace by the constant 0. In this network, $f_i = 1$, $f_j = f_k'$, and $f_l = f_k$. The network will realize the function $f_{\tilde{C}} = x_2 \vee \ldots \vee x_n$. Since $f_i = 1$, every fan-out gate of i will compute the same function as if the edge between i is removed. Therefore i has become redundant in $\tilde{C}$ and can be removed from the network. Similarly, since l now compute the function k, the node l can be replaced by the node k. This implies that gates j and l are no longer needed in the network, and can be removed. The result is a network that realizes the function OR_{n-1} and

contains 3 less gates than C. By the induction assumption, $|OR_{n-1}| \geq 3(n-2)$, so $|\tilde{C}| \geq 3(n-2)$. Therefore $|OR_n| = |C| \geq 3(n-2)+3 = 3(n-1)$. □

4.3.2.3 NAND Class

The table in Figure 4.24 gives the results of BESS on the functions from the NAND class. The algorithm was able to complete the search for functions up to 7 inputs with the largest network containing 11 gates. The search for an optimal network for the 8-input function was stopped after three days with the smallest network found thus far containing 13 gates.

Inputs	Cost of Optimal	Time for Completion	Search Tree Size
2	1	0 s	5
3	3	0 s	7
4	5	0 s	25
5	7	0 s	310
6	9	8 s	13,885
7	11	15.7 min	1,401,599
8*	13	3 days	295,212,881

Figure 4.24: NAND Class Results

Figure 4.25 gives the optimal networks for the NAND function when $n = 2,3,4$. The pattern that emerges from these networks is based on the combination of $(n-2)$ two-input AND networks with an additional NAND gate at the output connecting two disjoint AND networks. Since a two-input AND network requires two NAND2 gates, the cost of an optimal network for an n-input NAND function is $2(n-2)+1$.

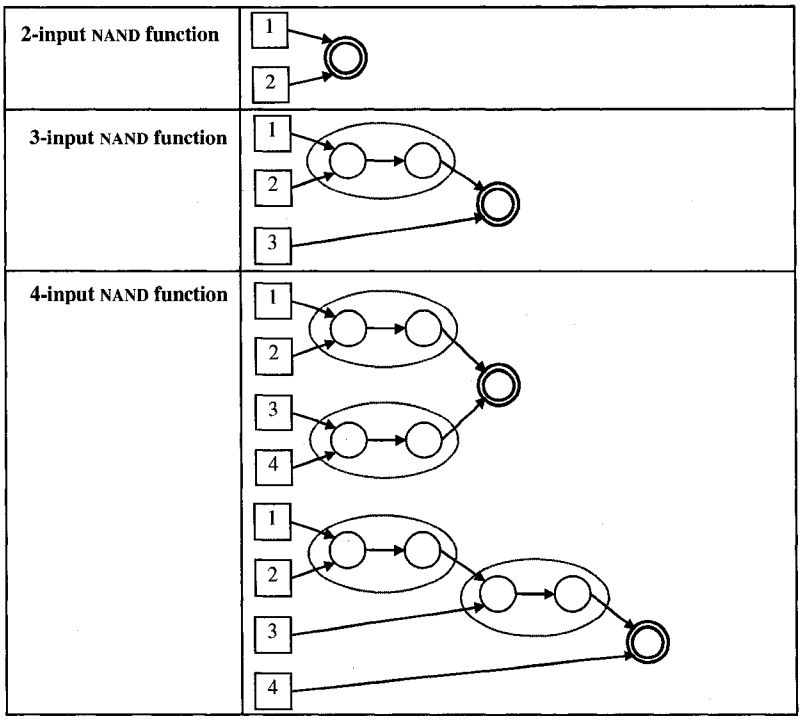


Figure 4.25: Optimal Networks for NAND Class

This construction gives an upper bound on the optimal cost of an n-input NAND network, $|\text{NAND}_n|$. The following theorem shows that this formula also gives a lower bound on the optimal cost.

Theorem 4.3. *The number of gates required in an optimal network for an n-input NAND function is at least* $2(n-2)+1$. $|\text{NAND}_n| \geq 2(n-2)+1$.

The proof of the AND lower bound can be used to prove this lower bound as well.

4.3.2.4 NOR Class

The table in Figure 4.26 gives the results of BESS on the functions from the NOR class. The algorithm was able to complete the search for functions up to 6 inputs with the largest network containing 16 gates. The search for an optimal network for the 7-input function was stopped after two days with the smallest network found thus far containing 19 gates.

Inputs	Cost of Optimal	Time for Completion	Search Tree Size
2	4	0 s	5
3	7	0 s	16
4	10	0 s	129
5	13	4 s	6,796
6	16	30.9 min	2,527,493
7*	19	3 days	301,275,147

Figure 4.26: NOR Class Results

Figure 4.27 gives the optimal networks for the NOR function when $n = 2,3,4$. Similar to the NAND class, the optimal network for an n-input NOR function uses OR networks. Here, the combination of an n-input OR network and an inverting NAND2 gate produces an n-input NOR network. Since an n-input OR circuit requires $3(n-1)$ NAND2 gates, the cost of an optimal network of an n-input NOR function is $3(n-1)+1$.

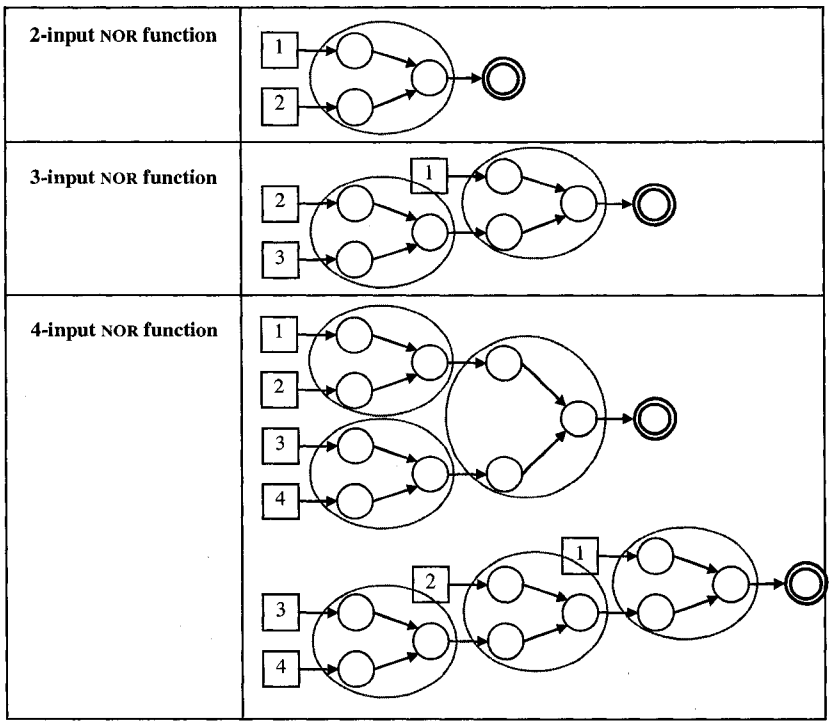


Figure 4.27: Optimal Networks for NOR Class

Once again, this construction gives an upper bound on the optimal cost of an n-input NOR network, $|\text{NOR}_n|$. The following shows that this formula also gives a lower bound on the optimal cost.

Theorem 4.4. *The number of gates required in an optimal network for an n-input NOR function is at least 3(n-1)+1.* $|NOR_n| \geq 3(n-1)+1$.

The proof of the OR lower bound can be used to prove this lower bound as well.

4.3.2.5 XOR Class

The table in Figure 4.28 gives the results of BESS on the functions from the XOR class. The algorithm was able to complete the search for functions up to 5 inputs with the largest network containing 16 gates.

Inputs	Cost of Optimal	Time for Completion	Search Tree Size
2	4	0 s	13
3	8	0 s	416
4	12	36 s	48,025
5	16	22 hrs	163,486,233

Figure 4.28: XOR Class Results

Figure 4.29 gives the optimal networks for the XOR function when $n = 2,3,4$. Assuming the pattern continues, these networks show that the optimal network for an n-input XOR function will be a combination of $(n - 1)$ two-input XOR circuits. Since a two-input XOR circuit requires four NAND gates, we conjecture that the cost of an optimal network for an n-input XOR function will be $4(n - 1)$.

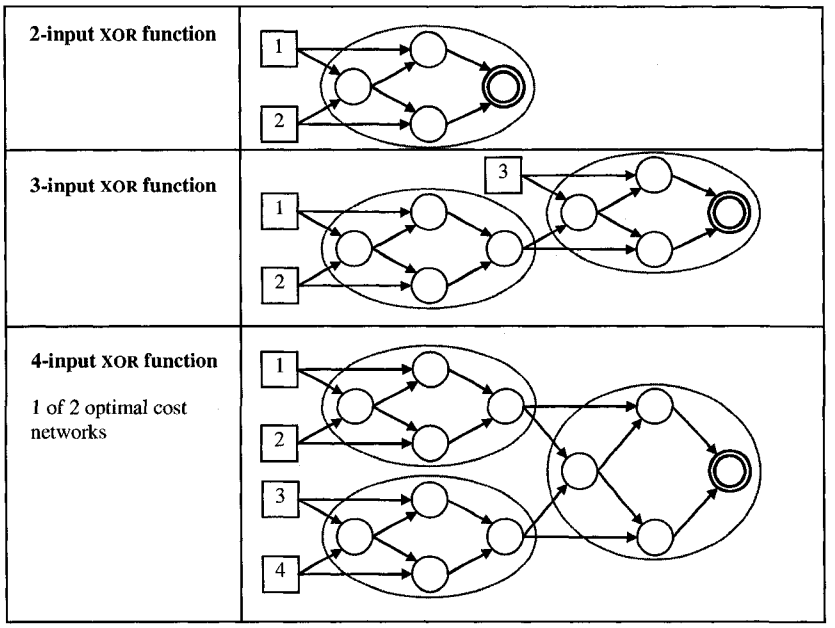


Figure 4.29: Optimal Networks for XOR Class

4.3.2.6 XNOR Class

The table in Figure 4.30 gives the results of BESS on the functions from the XNOR class. The algorithm was able to complete the search for functions up to 4 inputs with the largest network containing 13 gates. The search for an optimal network for the 5-input function was stopped after three days with the smallest network found thus far containing 17 gates.

Inputs	Cost of Optimal	Time for Completion	Search Tree Size
2	5	0 s	15
3	9	0 s	338
4	13	2.0 min	167,496
5*	17	3 days	223,605,759

Figure 4.30: XNOR Class Results

Figure 4.31 gives the optimal networks for the XNOR function when $n = 2,3,4$. Just like the NOR and NAND networks, the XNOR networks are created from a combination of XOR networks. Here, $(n-2)$ two-input XOR networks are combined with a two-input XNOR network to create an n-input XNOR network. Many optimal networks exist as n increases since the two-input XNOR network can appear anywhere within the basic structure. Based on this evaluation, we conjecture that the cost of an optimal network of an n-input XNOR function is $4(n-2)+5$.

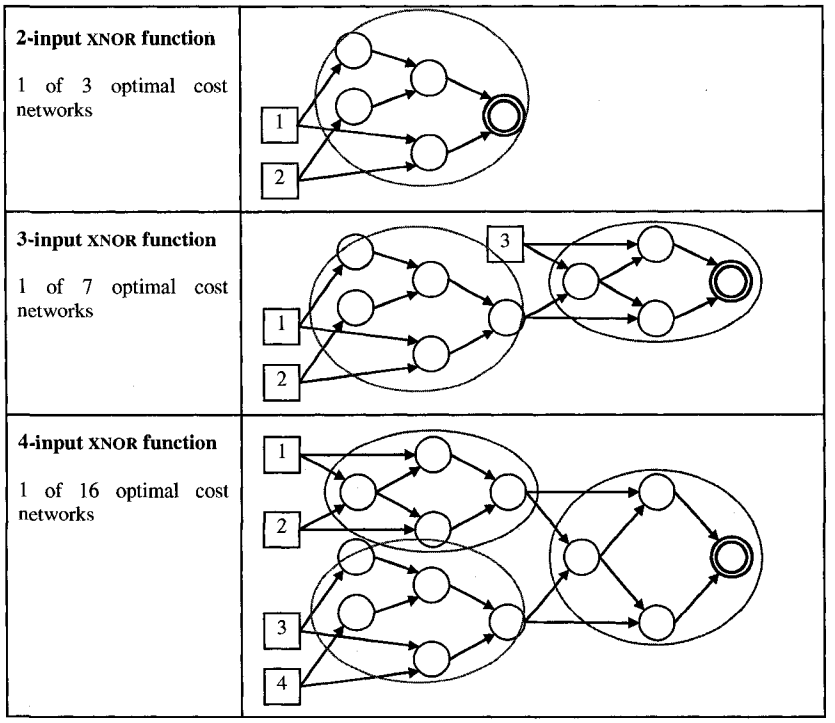


Figure 4.31: Optimal Networks for XNOR Class

4.3.2.7 MAJORITY Class

The table in Figure 4.32 gives the results of BESS on the functions from the MAJORITY class. The algorithm was able to complete the search for functions up to 5 inputs with the largest network containing 15 gates. The search for an optimal network for the 6-input function was stopped after three days with the smallest network found thus far containing 31 gates.

Inputs	Cost of Optimal	Time for Completion	Search Tree Size
2	2	0 s	5
3	6	0 s	51
4	8	0 s	425
5	15	38.3 hrs	170,359,437
6*	31	3 days	177,955,331

Figure 4.32: MAJORITY Class Results

Figure 4.33 gives the optimal networks for the MAJORITY function when $n = 2,3,4$. This class of networks is more difficult to analyze than the previous examples. No simple structure repeats for larger sized networks. Further discussion on the complexity of this class will be given Section 4.3.2.11.

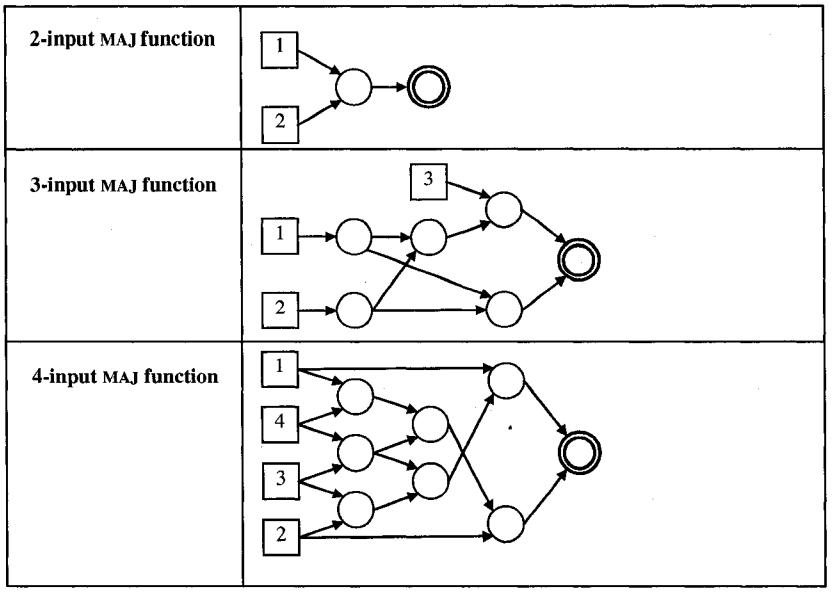


Figure 4.33: Optimal Networks for MAJORITY Class

4.3.2.8 Multiplexer Class

The table in Figure 4.34 gives the results of BESS on the functions from the MUX class. The inputs of the multiplexer can be divided into two groups: the variables and the selectors. The value of the selection inputs determine which variable is connected to the output. At least $s = \lceil \log_2 v \rceil$ selectors will be required for v variables. In the case when $s < \log_2 v$ multiple encodings may be possible. The number of possible encodings is provided in the column under the "Possible Functions" heading. Details of the algorithm are then provided for all of these possible functions.

The algorithm was able to complete the search for functions up to 6 inputs with the largest network containing 11 gates. The search for an optimal network for the 8-input functions was stopped after three days with the smallest network found thus far containing 18 gates.

Inputs	Variables	Selectors	Possible Functions	Cost of Optimal	Time for Completion	Search Tree Size
3	2	1	1	4	0 s	10
5	3	2	4	8 - 9	0 s	242 – 868
6	4	2	1	11	33 s	49,095
8*	5	3	56	18	3 days	320,642,269

Figure 4.34: MUX Class Results

Figure 4.35 gives the optimal networks for the MUX function when $n = 3,6$. These diagrams show that when $s = \log_2 v$ the optimal network for an n-input MULTIPLEXER function can be made from a v-input OR network with s additional NAND gates: Cost $= 3(v-1)+s$.

When the number of variables for the encoding falls between 2^{s-1} and 2^s, there are multiple ways that the encoding can be performed. Different encodings of the variables will produce different sized networks. Since a network can be created by taking the optimal network which encodes all 2^s variables and removing the unused variables, the optimal network will have cost $\leq (3(2^s-1)+s)-(2^s-v) = 2^{s+1}+s+v-3$. In the 3 variable case where n = 5, the cost of the network must be less than or equal to 10. The encoding (11, 01, 10) results in an optimal network of cost 8, while the other three encodings (11, 01, 00), (11, 10, 00), (01, 10, 00) produce networks of cost 9. This same formula implies that the cost of the network with 8 inputs (5 variables and 3 selectors) should have cost less than or equal to 31.

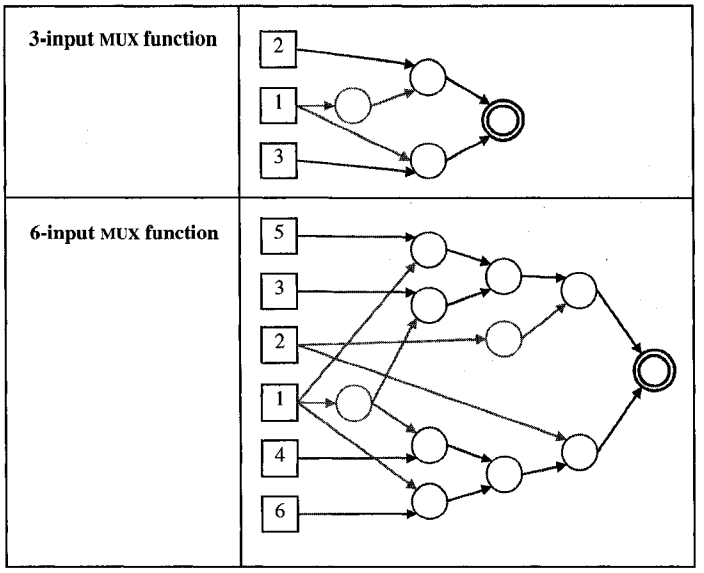


Figure 4.35: Optimal Networks for MUX Class

4.3.2.9 Threshold Functions

The table in Figure 4.36 gives the results of BESS on functions from the THRESHOLD class. A function from this class with n inputs and a threshold of k evaluates to true when at least k of the n inputs are true. These threshold functions include two classes of functions discussed previously. When k = 1 the threshold

function is simply the OR function on n inputs. When $k = n$ the threshold function is the AND function on n inputs. The results presented in the table are only for those functions on which the search was completed. For the threshold function, we were not able to obtain enough data to construct a formula for the cost of an optimal network based on the number of inputs to the function.

Inputs (n)	Threshold (k)	Cost of Optimal	Time for Completion	Search Tree Size
2	1	3	0 s	5
3	1	6	0 s	16
4	1	9	0 s	129
5	1	12	3 s	6,796
6	1	15	30 min	2,527,493
2	2	2	0 s	5
3	2	6	0 s	51
4	2	11	18 s	32,651
3	3	4	0 s	7
4	3	8	0 s	425
4	4	6	0 s	25
5	4	13	36 min	3,289,916
5	5	8	0 s	310
6	6	10	9 s	13,885
7	7	12	16 min	1,401,599

Figure 4.36: THRESHOLD Class Results

4.3.2.10 Adder Class

The table in Figure 4.37 gives the results of BESS on the functions from the ADDER class. The algorithm was able to complete the search for functions up to 4 inputs with the largest network containing 14 gates. The search for an optimal network for the 5-input function was stopped after three days with the smallest network found thus far containing 18 gates.

Adder	Inputs	Outputs	Cost of Optimal	Time for Completion	Search Tree Size
Half	2	2	5	0 s	48
Full	3	2	9	0 s	1,212
2 bit without carry	4	3	14	81 s	121,951
2 bit with carry*	5	3	18	3 days	179,645,168

Figure 4.37: ADDER Class Results

Figure 4.38 gives the optimal networks for the first three adders.

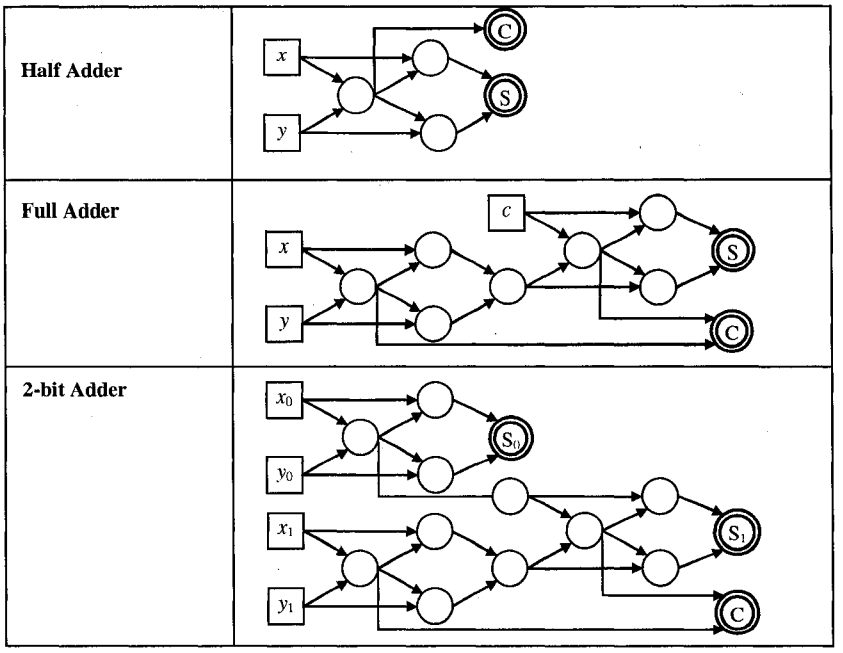


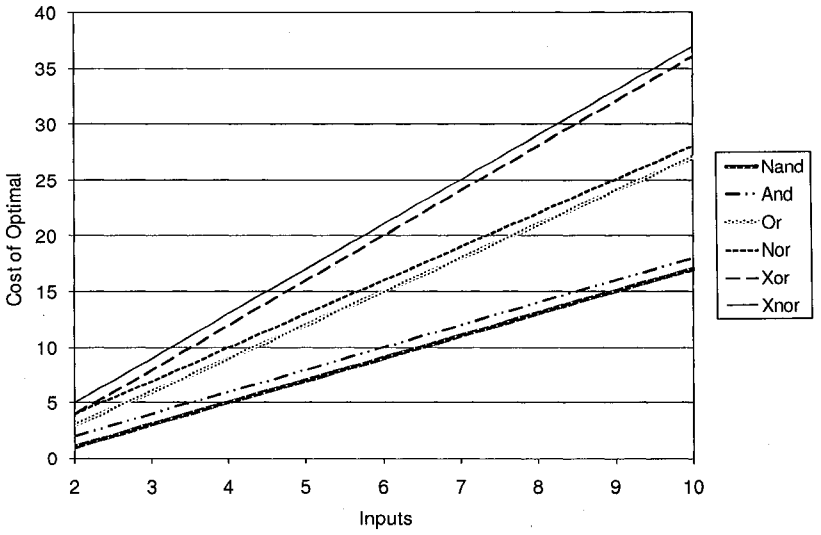
Figure 4.38: Optimal Networks for ADDER Class

The optimal cost networks found by the algorithm give the basic structure expected for the adders. The full adder is created from a combination of two half adders, while the 2-bit adder is created from a combination of a full and half adder. Assuming this pattern continues for larger functions, larger adders can be created by simply adding full adders to the network of smaller size. Therefore the cost an optimal network for an n bit adder is $9n$.

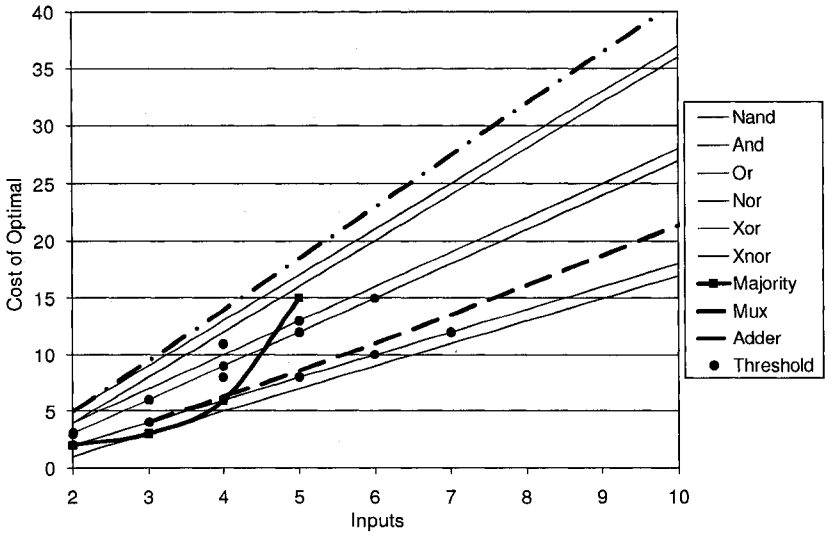
The work by Lai and Muroga [Lai 79] performs a similar analysis for the ADDER class using NOR gates. Their work shows that when no fan-in, fan-out or level restriction is placed on the network then an optimal network of NOR gates for an n bit adder requires $7n + 1$ gates. In this work they show that the optimal network for the n bit adder will be composed of one-bit adder modules similar to the modular construction we have seen here.

4.3.2.11 Analysis of Results

Using the class results presented the function classes can be categorized based on the size of the optimal solutions and the difficulty of the search for finding these optimal solutions. The first graph in Figure 4.39 shows a plot of the cost formulas for the optimal networks for six of the function classes where a cost formula was obtained based on the modular structure of the networks. From this graph an ordering of these classes can easily be obtained. This ordering indicates that the XOR and XNOR classes are more difficult to implement with NAND2 gates than the OR and NOR classes which in turn require more gates to implement than the AND and NAND classes.



(a) Function classes where a cost formula was obtained based on a modular structure



(b) All Function classes

Figure 4.39: Cost Formula for Optimal Networks for Function Classes

If data from the other classes of functions are plotted along with these 6 classes, then a comparison with these classes can be made. The second graph in Figure 4.39 indicates that the multiplexer formula falls between the AND/NAND classes and the OR/NOR classes in difficulty. The ADDER formula follows the XOR and XNOR classes. This is to be expected after looking at the structure of the networks given in Figure 4.38.

Using the data obtained by the algorithm on the majority and threshold functions, a similar analysis can be performed. The plot of the majority function, in relation to the other functions, gives an indication of

why a formula was not easily found for this class. The data do not follow the linear pattern found in other function classes, implying that this class of functions is inherently different than the others.

In the threshold class the two sub-classes of the AND and OR functions are seen with the data points that follow the data lines for these two classes. There are not enough additional data points from this function class to provide further analysis, however it does appear that these functions will follow a linear pattern similar to the AND and OR classes.

The networks presented in the previous sections show that most of the function classes possess a modular pattern. Larger networks are created by cascading smaller networks of the same class. The exception to this pattern is the majority function. The modular pattern results in a linear function which describes the cost of the network. The size of the repetitive portion of the network gives the slope of the line, and therefore indicates the complexity of this class.

4.3.3 Optimal Results for Benchmarks

In this section, results of the exact synthesis algorithm on the set of benchmark functions are presented. These data allow us to evaluate the algorithm on large functions with varying number of inputs and outputs. Figure 4.40 gives a table of results for the benchmark functions. Here the function along with its description (number of inputs and number of outputs) are given in the first three columns. Following that, the results of the algorithm are presented for each function. This includes the cost of the optimal network, the time it took for the algorithm to complete and the size of the search tree.

Many of the functions are too large for the algorithm to complete in the allotted time of three hours. These functions are indicated with a * by their name. The details of the search are included for the functions as well the cost of the smallest network found during the search. In addition to these functions, on still more functions the algorithm was not able to find an initial network at all. These functions are indicated with a $^{+}$ by their name. The cost given for these functions is the size of the network when the algorithm terminated due to the size of the search tree.

Name	Inputs	Outputs	Cost	Time	Search Tree Size	Avg. Height	Avg. Width
2bit_adder*	5	3	34	3 hrs	5,032,408	83.620	2.809
4bit_adder$^+$	9	5	166				
alu2$^+$	10	6	100				
b1	3	3	10	0 s	682	11.407	2.450
C17	5	2	6	0 s	44	9.000	2.389
cc$^+$	21	15	53				
cm138a*	6	8	79	3.0 hrs	1,413,961	148.674	2.684
cm151a$^+$	12	2	137				
cm152a*	11	1	41	3.0 hrs	5,364,285	104.689	2.589
cm162a$^+$	14	5	75				
cm163a$^+$	16	5	23				
cm42a*	4	10	65	3.0 hrs	1,903,469	115.828	3.274
cm82a*	4	3	59	3.0 hrs	5,020,141	130.785	1.930
cm85a*	11	3	193	3.0 hrs	26,098	908.972	5.145
cmb*	16	4	120	3.0 hrs	537,475	558.258	2.871
cu$^+$	14	11	77				
decod*	5	16	138	3.0 hrs	166,357	237.499	3.042
f51m$^+$	8	8	277				
majority	5	1	9	2 s	6,886	14.567	2.419
muroga	3	1	7	0 s	110	7.328	2.224
oai22	4	1	7	0 s	99	6.410	2.450
partialMux	6	1	12	18 s	35,801	17.453	2.562
pcle$^+$	19	9	38				
pm1$^+$	16	13	33				
sct$^+$	19	13	66				
small	3	1	2	0 s	7	3.250	2.000
tcon$^+$	17	8	30				
x2*	10	7	130	3.0 hrs	761,098	898.725	2.896
z4ml*	7	4	200	3.0 hrs	224,283	592.582	3.211

Figure 4.40: MCNC Benchmark Results

The results presented here give us an idea of how the algorithm would perform on larger more varied functions. These results show that once the combination of inputs and outputs becomes larger than 8 an optimal solution can no longer be found in the allotted time. The cost of these networks confirms what was suggested by the previous results. Networks requiring more than 16 gates can not be completed in less than 3 hours no matter the number of inputs or outputs. Once the combination of inputs and outputs becomes large the size of the optimal network must also increase, thus the likelihood reduces that the algorithm will be able to find an initial network before the search space becomes too large.

4.4 Search Tree Analysis

In Section 3.6 the search space of the algorithm was discussed and bounds on the size of the search tree produced by the algorithm were given. This section will use the experimental results of the algorithm to evaluate these bounds. Additional run-time analysis will then be provided for the algorithm based on these results.

4.4.1 Search Tree Size

As described earlier, the total number of nodes in a search tree is defined by $\frac{w^{h+1}-1}{w-1}$, where w is the width of a branch in the search tree and h is the height of the tree. The width of a branch in the search tree is determined by the number of options for covering an uncovered minterm. The height of a path in the search tree is the number of coverings that are performed before a complete network is found or the current partial network exceeds the cost of the current minimum. Therefore determining the maximum width and height of the search tree for a given function will determine an upper bound for the size of the tree.

4.4.1.1 Branch Width

In Section 3.6.3.1 two bounds were given for the width of a branch in the search tree. The first was based on the number of inputs n of the function and the maximum number of nodes that exist in a partial network, G_{max}. In this case $w < G_{max} + n$. The second bound was based on the number of n-input Boolean function that could cover a minterm. In this case $w < 2^{2^n-1}$.

Using experimental data these bounds can now be evaluated by comparing the actual width of the branches in the search tree to the value of these bounds. The table of results in Figure 4.41 gives the details of the branch width from the search trees that were produced when the algorithm was run on functions from the P-equivalence class sets of 2, 3, and 4 inputs. These results show that the average branch width for all functions is between 2 and 3 while the maximum branch width seen for any function in the sets is 21. Comparing these experimental values with the bounds obtained in Section 3.6 shows that both bounds represent significant over approximations of the maximum branch width: by more than twice in the best case and by almost 25 times in the worst case.

Inputs	Functions	Search Tree Size			Branch Width			$G_{max} + n$	2^{2^n-1}
		Min	Avg.	Max	Min	Avg.	Max		
2	8	5	8	1	2	2.0	2	8	18
3	68	7	96	1,809	1	2.2	8	20	119
4	3904	9	47,843	9,548,355	0	2.4	21	40	540

Figure 4.41: Branch Width Data on P-Equivalence Class Functions

One characteristic of the algorithm which contributes to this low average value for the branch width is the gate fan-in bound. Since the fan-in is limited to two inputs, the number of coverings possible for a node which has a complete fan-in set will never exceed 2. The more nodes with complete fan-in sets which require covering, the more the average branchwidth will be pulled down to 2.

Some of the algorithmic improvements made to the algorithm also effect the width of the branches in the search tree. The first is the minterm selection heuristic. One of the goals for this heuristic is to minimize the width of the branch. This is accomplished by choosing the minterm for covering which has the smallest connectible set. Because of this selection, the width of most branches will be fairly small,

much smaller than the maximum value obtained with the bounds described earlier. The second algorithmic improvement that can effect the width of a branch in the search tree is the use of structural implications. By performing these implications as part of a covering rather than as an individual covering, almost all of the branches with the width of 1 will be eliminated. By combining these two improvements the average width of a branch will be kept between 2 and 3.

While the width bounds provided in Section 3.6 do give an upper bound on the width of a branch in the search tree, they do not give a good indication of what the actual size will be and therefore will provide a poor indication of the size of the search tree. A better estimate for the width of a branch in the search tree can be found by considering the average branch width from the experiments. The graphs provided in Figure 4.42 compare the average branch width of the search tree for numerous functions with respect to the number of inputs in the function and the cost of the optimal network. These graphs show that that the average width of a branch remains under three even for functions with more than 4 inputs and for functions with cost as high as 16. Comparing these two graphs, it appears that the cost of the optimal network is the better indication of the average branch width. The line through the data shown in the graph provides a trend for the average width based on the size of the optimal network. Using this equation in the search tree equation above, the size of the search tree can be approximated by $\frac{(.04c+2)^{h+1}-1}{.04c+1}$, where c is the cost of the optimal network.

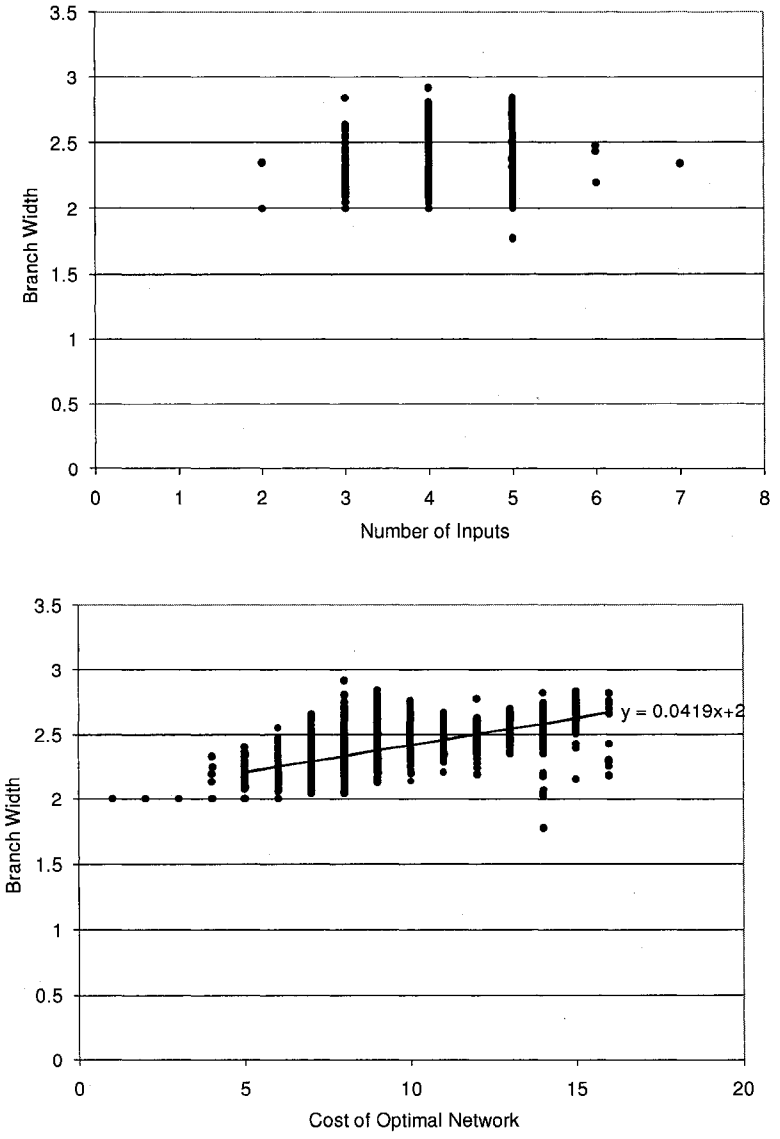


Figure 4.42: Branch Width as a Function of Input Size and Network Cost

4.4.1.2 Search Tree Height

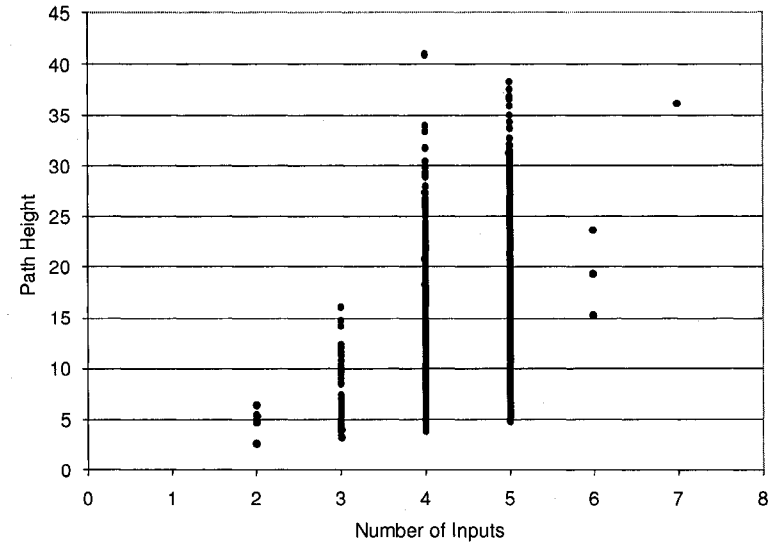
In Section 3.6 an upper bound for the height of a search tree was given based on the assumption that every minterm in every node may need to be covered by an individual branching step before the network is covered. This gave the bound $h < G_{max}\left(2^N - 1\right)$ for the height of the search tree. While this provides an upper bound on the height of the search tree, it will not be necessary to perform a covering for every on-set minterm of every node in the network. Many minterms will be covered as a result of functional and structural implications performed after a single covering is made.

Once again the results of the algorithm on the P-equivalence class functions with 2, 3, and 4 inputs can be used to provide data on the actual height of the search tree. The table in Figure 4.43 provides the details of these results. This table shows that the bound provided in Section 3.6 over approximates the maximum path height by 2 to 6 times the actual path height.

Inputs	Functions	Search Tree Size			Path Height			$G_{max}(2^N-1)$
		Min	Avg.	Max	Min	Avg.	Max	
2	8	5	8	1	2	3.29	7	18
3	68	7	96	1,809	2	6.77	30	119
4	3904	9	47,843	9,548,355	2	13.31	72	540

Figure 4.43: Path Height Data on P-Equivalence Class Functions

A better estimate of the average height of the search tree can be obtained using the results of the experiments. The graphs in Figure 4.44 compare the average height of a path in the search tree with the number of inputs in the network on the left and the optimal cost of the network on the right. Again the cost of the optimal network gives a better indication of the average path height. The line through the data in this graph is an exponential function which provides a trend for the average height based on the size of the optimal network. Using this equation in the search tree equation gives an approximation of the size of the search tree of $\frac{(.04c+2)^{2.8e^{0.16c}+1}-1}{.04c+1}$, where c is the cost of the optimal network.



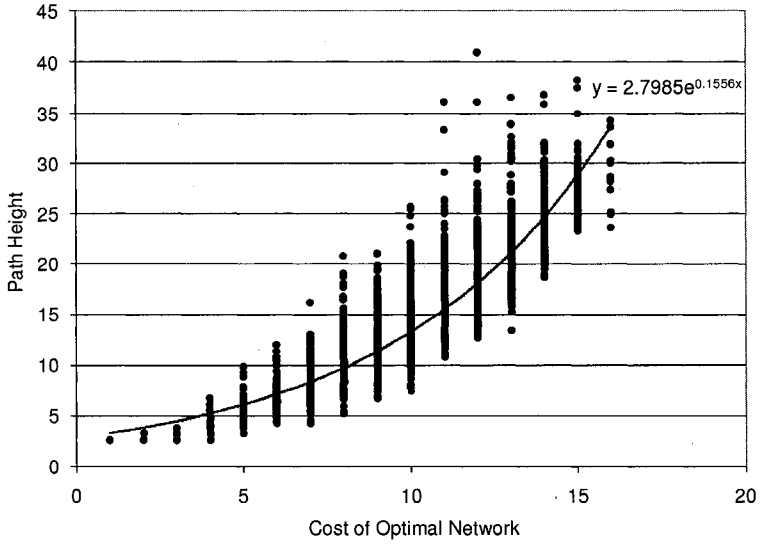


Figure 4.44: Average Path Height as a Function of Input Size and Network Cost

4.4.1.3 Search Tree Size

Now that both the height and width of the search tree have been analyzed based on experimental results, the approximations of the search tree size can be compared to the actual size of the search trees to determine how well the equations model the actual data. The two equations obtained in Section 3.6 that provide upper bounds on the size of the search tree were $\frac{(n+G_{max})^{G_{max}(2^n-1)+1}-1}{n+G_{max}-1} = O\left((n+G_{max})^{2^n G_{max}}\right)$ and $\frac{\left(2^{2^n-1}\right)^{2^{2^n}(2^n-1)+1}-1}{2^{2^n-1}-1} = O\left(2^{2^{2^n}}\right)$. In this section experimental evidence has been used to provide a better

approximation for the size of the search tree: $\frac{(.04c+2)^{2.8e^{0.16c}+1}-1}{.04c+1}=O\left(c^{2^c}\right)$. The graph on the left in Figure 4.45 provides a comparison of this new model with the actual data obtained from the various P-equivalence and functions classes organized according to the cost of the optimal network. This graph shows that the search tree size appears to follow a greater than exponential trend which is bounded above by this approximation equation.

The analysis thus far has shown that the cost of the optimal network provides the best indication of the size of the search tree. Therefore we can use this property to find a model directly from the search tree data. The second graph in Figure 4.45 shows the average search tree size as a function of the number of gates in the optimal network. These data are modeled by a greater than exponential function given by the dotted line in the graph. This model gives an indication that the size of the search tree for the average case given the cost of the optimal network.

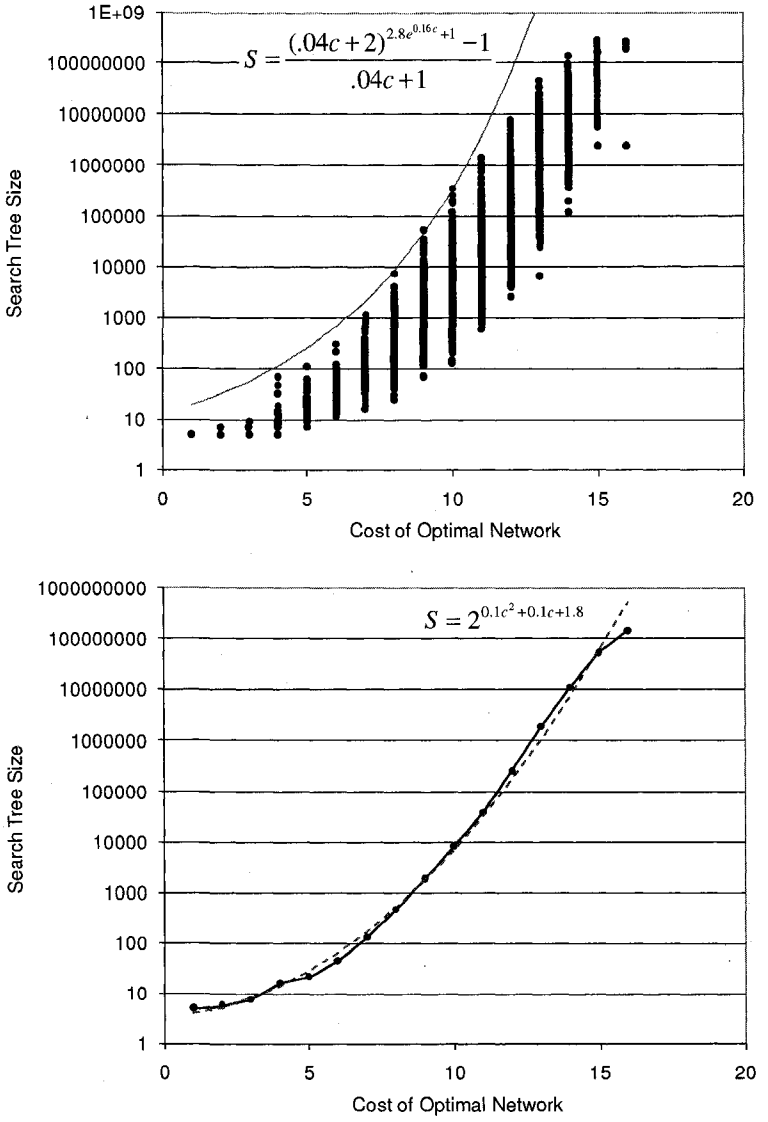


Figure 4.45: Search Tree Analysis Based on the Cost of the Optimal Network

4.4.2 Search Space Analysis using Function Classes

The function classes can also provide a view of the search space. Since the optimal solution for every function in a class will be constructed from the same pattern of nodes, the search tree produced by the algorithm on these classes should follow a pattern as well. The table and graph in Figure 4.46 give the size of the search tree for seven NAND functions.

Input	Minimum Cost	Search Tree Size	Branches
2	1	5	2
3	3	7	3
4	5	17	8
5	7	91	40
6	9	1,724	626
7	11	69,590	23,525
8	13	4,448,537	1,464,769

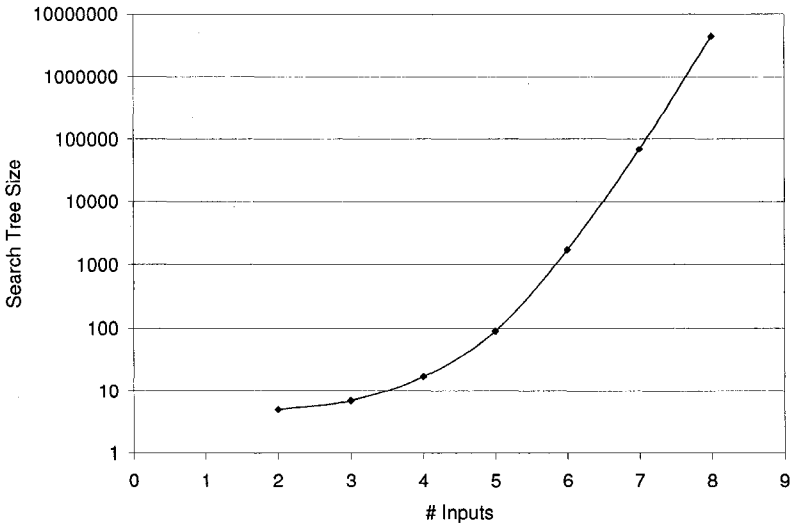


Figure 4.46: Search Tree Size for NAND functions

These points fit the equation $S(n) = 2^{1.5^n}$. Once again a double exponential model for the size of the search tree appears. The algorithm begins the search for an optimal network of an n-input NAND function by performing the exact same steps as performed in the search for an optimal network of an $(n - 1)$-input NAND function. Within this search, the optimal network is found. The optimal network is always the first network found for this class of functions. The remainder of the search space is used to prove the optimality of the network. Within this search, the search for an m-input NAND network is repeated for every $m < n$.

4.4.3 Search Space Analysis Conclusions

The results of both search space analyses provide a general idea of the complexity of the algorithm. First it was discovered that the cost of the optimal solution provides the best indication for the size of the search tree produced by the algorithm. It was discovered that the size of the search tree is double exponential in the cost of the optimal solution. This type of growth explains the sharp cutoff that occurred between the functions that the algorithm was able to complete and those which it was not in Section 4.3. Networks with cost between 13 and 16 provide the boundary of the algorithm. The algorithm was able to complete the search providing networks with cost up to 16, while in some cases the algorithm was not able

to complete the search for networks with cost as little as 13. Using the search tree approximation, the size of an average search tree for a network requiring 13 gates should be approximately 1.1×10^6. When the optimal network increases to 16 gates, we expect the average size of the search tree to increase to 5.4×10^8.

4.5 Conclusions

In this chapter experimental results were provided for the algorithm presented in Chapter 3. First, experimental justifications for the algorithmic improvements were given. Next, optimal results for several groups of Boolean functions were provided. The P- and NPN-equivalence classes provided a database of optimal networks for functions with 5 or fewer inputs. The function classes gave results on larger functions and through analysis of the structures of the networks gave formulas for optimal networks within certain classes. The benchmark functions gave even larger functions which allowed us to evaluate the algorithm on a more varied group of functions. Here evaluations of the algorithm with respect to network size, input size and output size could be made. This chapter concluded with an analysis of the search using experimental results. An approximation for the size of the search tree in terms of optimal network cost was obtained.

Chapter 5

Variations on Optimal Synthesis

The optimal synthesis algorithm, BESS, used to provide the results given in Chapter 4 used a very specific set of constraints for the networks generated. One of the advantages of BESS is that many of these constraints can easily be changed. In this chapter we will explore some of the variations that are possible. For each variation we will describe how the algorithm needs to be changed to produce the variation as well as how the variation effects the optimal network results and the performance of the algorithm. Then we will provide results of the algorithm on select functions. Finally, we will compare these results to similar results found in the literature.

5.1 Summary of Previous Results

Since the results from Chapter 4 will be used repeatedly for comparison purposes, we summarize those results here. Figure 5.1 gives a summary of the results of BESS on the representative function sets for inputs 2, 3, and 4 and a summary of the results of the algorithm on eight classes of functions. The algorithm was allowed 3 hours to complete each function. For the classes, functions on an increasing number of inputs were evaluated until the search could no longer be completed in the allotted time. The number given in the second column of table (b) gives the largest function on which the algorithm found and proved an optimal network.

Inputs	Functions	Complete	Cost	Search Tree Size	Avg. Height	Avg. Width
2	8	8	24	58	3.25	2.00
3	68	68	405	6,013	6.71	2.20
4	3,904	3,904	37,936	169,979,553	12.95	2.41

(a) Representative Functions

Function Class	Max Input Completed	Cost	Search Tree Size	Avg. Height	Avg. Width
AND	7	42	1,415,831	12.90	2.10
NAND	7	36	1,415,831	12.90	2.10
OR	6	45	2,534,439	10.37	2.25
NOR	6	50	2,534,439	10.37	2.25
XOR	4	24	48,454	16.13	2.38
XNOR	4	27	167,849	13.74	2.50
MAJ	5	31	170,359,918	12.64	2.29
MUX	6	50	51,653	11.04	2.40

(b) Classes of Functions

Figure 5.1: Results of BESS on Representative Functions

In both tables, the last 4 columns give the information from the search for all functions in the class. First, the sum of the costs for the optimal networks of all functions in the class is given. Next the total size of the search trees for all functions is given. The last two rows show the average height of a path in the search trees and the average width of a branch in the search trees.

5.2 Fan-in and Fan-out Restrictions

Restrictions on both the fan-in and fan-out of the gate nodes can be enforced. In the original version a fan-in restriction of 2 is enforced on the gate nodes. The same type of restriction can be placed on the fan-out of the node as well. In this section different fan-in and fan-out restrictions are placed on the gate nodes. In addition, the case when no restriction is used for the fan-in and fan-out sets will be considered.

5.2.1 Variation: Fan-out = 1

In our first variation we will restrict the fan-out of the gate nodes to 1. This fan-out restriction limits the structure of the resulting networks. This results in a simpler synthesis problem since the reuse of gate nodes within the network is no longer possible.

5.2.1.1 Changes to the Algorithm

In order to accommodate the new fan-out restriction, the computation of the connectible set must be changed . In the original version there are four types of nodes that can be used to cover an uncovered on-set minterm of a given node i: (1) primary input nodes, (2) existing gate nodes that already exist as an input

of i, (3) existing gate nodes that are not an input of i, and (4) new gate nodes. The fan-out restriction limits the type of covering nodes down to only three: (1) primary input nodes, (2) existing gate nodes that already exist as an input of i, (3) existing gate nodes that are not an input of i, (3) new gate nodes. Existing gate nodes that are not connected as an input of node i can not be used since the fan-out of these nodes would become greater than one if used to cover in this situation. Thus the `UpdateConnectibleSet` procedure will only need to consider nodes of types (1), (2), and (4) as options for inclusion in a node's connectible set. The result of this procedural change is that at most $(n + 2)$ nodes (where n is the number of input nodes) need to be searched each time the connectible set is updated rather than $(n + g)$ nodes (where g is the number of gate nodes in the network) as in the original version.

With this simple change made, the algorithm will now optimally synthesize Boolean functions into fan-out free networks. However, one additional modification can be made to improve the efficiency of the search. This modification removes the global functional implications used in the original version of the algorithm. All functional implications made based on these structures are no longer possible since reconvergent fan-out cannot exist if the base node has only a single fan-out. The algorithm can save time by skipping the search for these structures altogether.

5.2.1.2 Experimental Results

Two tables of results are given using this variation of the algorithm where the fan-out set of the gate nodes is limited to 1. Table (a) in Figure 5.2 shows the results of this variation on the set of representative functions for the P-equivalence classes on 2, 3, and 4 inputs. The details of the search are given in columns 4 – 7. Table (b) in Figure 5.2 shows the results of this variation on the classes of functions. A comparison of these results with results from the original version is given in the graphs of Figure 5.3. All functions from the three P-equivalence classes and 8 function classes are plotted along the x-axis according to their cost and search tree size, respectively. The first graph compares the cost of the optimal network for each function while the second graph compares the size of the search tree for each function.

Inputs	Functions	Complete	Cost	Search Tree Size	Avg. Height	Avg. Width
2	8	8	25	50	3.02	2.00
3	68	68	449	11,531	6.14	2.02
4	3,904	3,904	47,329	609,213,354	14.31	2.06

(a) Representative Functions

Function Class	Max Input Completed	Cost	Search Tree Size	Avg. Height	Avg. Width
AND	8	56	973,982	19.80	2.09
NAND	8	49	973,982	19.80	2.09
OR	6	63	445,634	14.25	2.01
NOR	6	69	445,634	14.25	2.01
XOR	4	42	11,588,443	17.30	2.07
XNOR	4	41	9,429,410	16.30	2.05
MAJ	5	34	533,873	11.77	2.04
MUX	6	56	7,832	12.47	2.05

(b) Classes of Functions

Figure 5.2: Results of Fan-out = 1

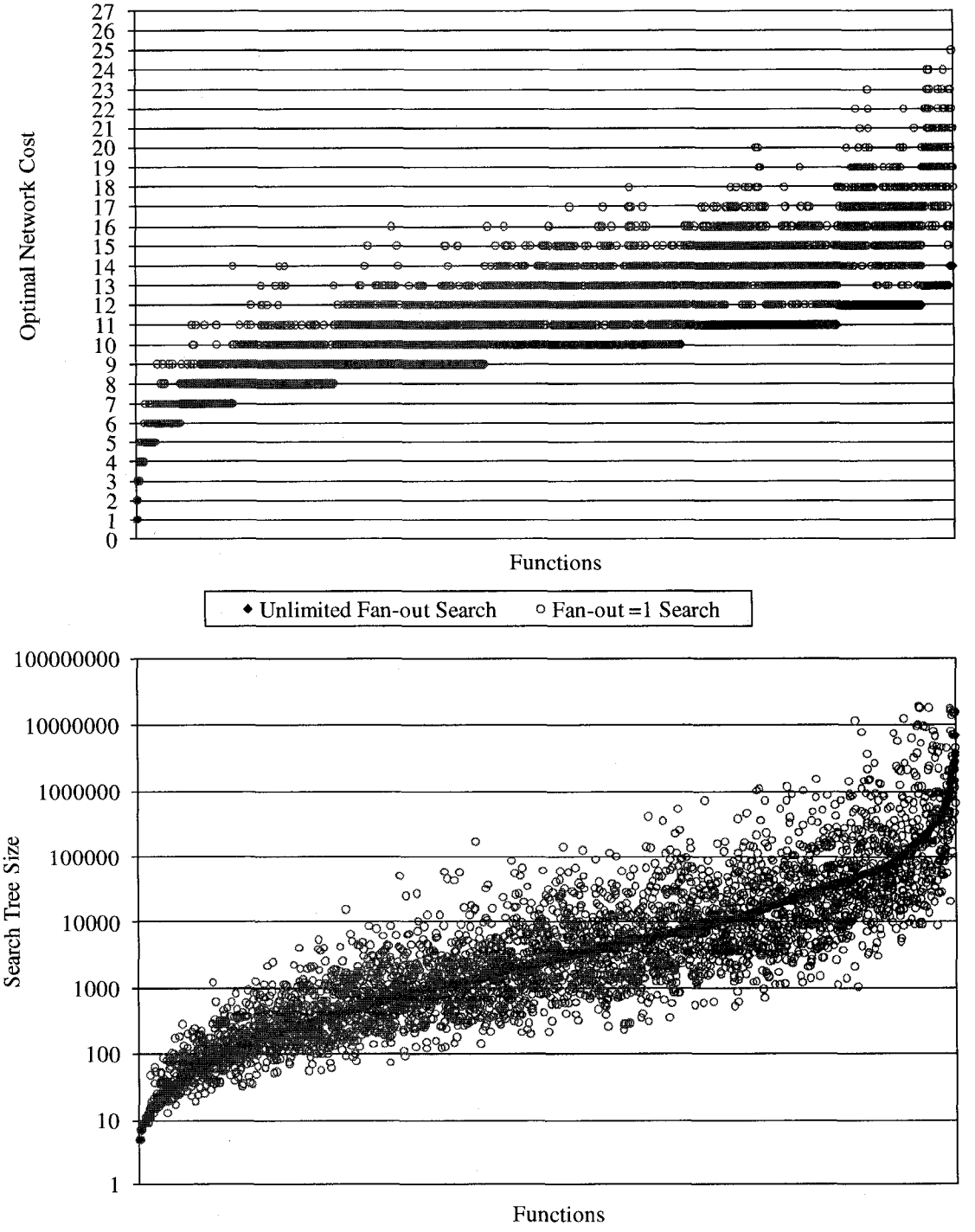


Figure 5.3: Graphs of Results when Fan-out = 1

The most obvious effect that this fan-out restriction has is on the size of the optimal networks. By removing the possibility for interconnections between the gate nodes in a network, larger networks must now be created for functions which previously required gate nodes of larger fan-out. This includes functions from the XOR and XNOR classes where the networks increased by as many as 11 gates. Some optimal networks will remain the same however as these networks originally had a fan-out free structure. This is true for the functions from the AND, NAND, OR, and NOR classes. From the results we found that out of the 3,980 representative functions, 3,421 had optimal fan-out free networks with larger cost than the optimal network found by the original version. Of these networks, the largest increase in cost was 92% (11 gates) while the average was 27% (2.8 gates).

The changes made to the algorithm to produce these networks effect the search that is performed. Since existing gate nodes can no longer be used in the covering, the options for covering are greatly reduced from $(n + g)$ to $(n + 2)$. This reduction has two effects on the search. The first is that the search time for the connectible set is reduced as fewer nodes must be searched. This reduction, along with a reduction in the time needed to propagate functional implications through the network, provide a reduction in the average time spent on a single branch in the search tree. The experimental results show that the average number of branches of the search tree that can be explored in 1 second is increased from 815 to 1,186.

The second effect that a reduction in the covering options has on the search is that the size of the connectible sets is reduced. This effects the width of the branches in the search tree. The experimental results show that the average width of a branch in the search tree is reduced 14%.

With the reduction in the width of the branches of the search tree, we would expect to see a reduction in the total size of the search tree. This is not always the case however. The second graph in Figure 5.3 shows that for about half of the functions a larger search must be completed. The increased size of the network plays a role in this increase. When the number of nodes is not increased, a reduction in the search tree size is seen. This is the case of the functions classes AND, OR, NAND, and NOR. However, when the number of nodes increases in the network, more on-set minterms must be covered in order for the network to be complete. Therefore we expect when the cost of the minimal network increases the size of the search tree will also increase. This is true for the function classes XOR and XNOR.

5.2.2 Variation: Fan-in = 3, Fan-out = 3

In the previous variation, the algorithm was changed to limit the fan-out of the gate nodes. The variation described here shows that any restriction can be placed on the fan-in and fan-out of the gate nodes. We increase both the fan-in and fan-out restrictions from the previous variation to 3.

5.2.2.1 Changes to the Algorithm

One change that must be made to the algorithm when any fan-in or fan-out restriction is used occurs within the `UpdateConnectibleSet` procedure. When finding the connectible set of a gate node *i*, the procedure must guarantee that using a node *l* from the connectible set for covering will not increase the size of the fan-in set of *i* and the fan-out set of *l* past the restrictions.

A change must also be made to the `PropagateFunctionalImplications` procedure. When the fan-in limit was 2, the local function of a gate node (*i, j, k*) was $f_i = f_j' \vee f_k'$. Based on this relationship the functional implications followed:

- Forward implications :

  $OFF_j \rightarrow ON_i$ and $OFF_k \rightarrow ON_i$

  $ON_j \wedge ON_k \rightarrow OFF_i$

- Backward implications:

  $OFF_i \rightarrow ON_j$ and $OFF_i \rightarrow ON_k$

  $ON_i \wedge ON_j \rightarrow OFF_k$ and $ON_i \wedge ON_k \rightarrow OFF_j$

Now that three nodes are allowed in the fan-in set of a gate node, the local function of a gate node (*i, j, k, l*) is $f_i = f_j' \vee f_k' \vee f_l'$. Therefore the functional implications should be extended to:

- Forward implications :

  $OFF_j \rightarrow ON_i$ and $OFF_k \rightarrow ON_i$ and $OFF_l \rightarrow ON_i$

  $ON_j \wedge ON_k \wedge ON_l \rightarrow OFF_i$

- Backward implications:

  $OFF_i \rightarrow ON_j$ and $OFF_i \rightarrow ON_k$ and $OFF_i \rightarrow ON_l$

  $ON_i \wedge ON_k \wedge ON_l \rightarrow OFF_j$ and $ON_i \wedge ON_j \wedge ON_l \rightarrow OFF_k$ and $ON_i \wedge ON_j \wedge ON_k \rightarrow OFF_l$

These same implications can be extended for any finite value assigned as the fan-in restriction. The on-set of a gate node is updated any time the off-set of one of its fan-in nodes is changed. The off-set of a node is only updated when the fan-in limit of the gate node has been reached.

All pruning techniques, global functional implications, and structural implications can remain unchanged from the original version since the properties of the network that allowed for these techniques to be used has remained unchanged as a result of the fan-in and fan-out restrictions.

5.2.2.2 Experimental Results

The two tables of results using this variation of the algorithm are given here. Table (a) in Figure 5.4 shows the result of this variation on the set of representative functions for the P-equivalence classes on 2, 3, and 4 inputs. Table (b) shows the results of this variation on eight of the classes of functions. Details of

the optimal cost and size of the search tree for individual functions are given in the graphs in Figure 5.5. These graphs compare this version of the algorithm to the previous version from Section 5.2.1 where the fan-in sets are limited to 2 nodes and the fan-out sets are limited to 1.

Inputs	Functions	Complete	Cost	Search Tree Size	Avg. Height	Avg. Width
2	8	8	24	58	3.25	2.00
3	68	68	328	3,467	6.50	2.18
4	3,904	3,904	29,764	793,162,813	13.82	2.45

(a) Representative Functions

Function Class	Max Input Completed	Cost	Search Tree Size	Avg. Height	Avg. Width
AND	8	32	439,738	18.28	2.11
NAND	8	25	439,738	18.28	2.11
OR	6	33	100,014	8.16	2.29
NOR	6	38	100,014	8.16	2.29
XOR	4	22	14,506,324	15.68	2.35
XNOR	4	23	550,491	13.56	2.43
MAJ	5	23	7,079,041	11.02	2.24
MUX	6	25	2,321	10.41	2.63

(b) Classes of Functions

Figure 5.4: Results of Fan-out = 3, Fan-in = 3 Variation

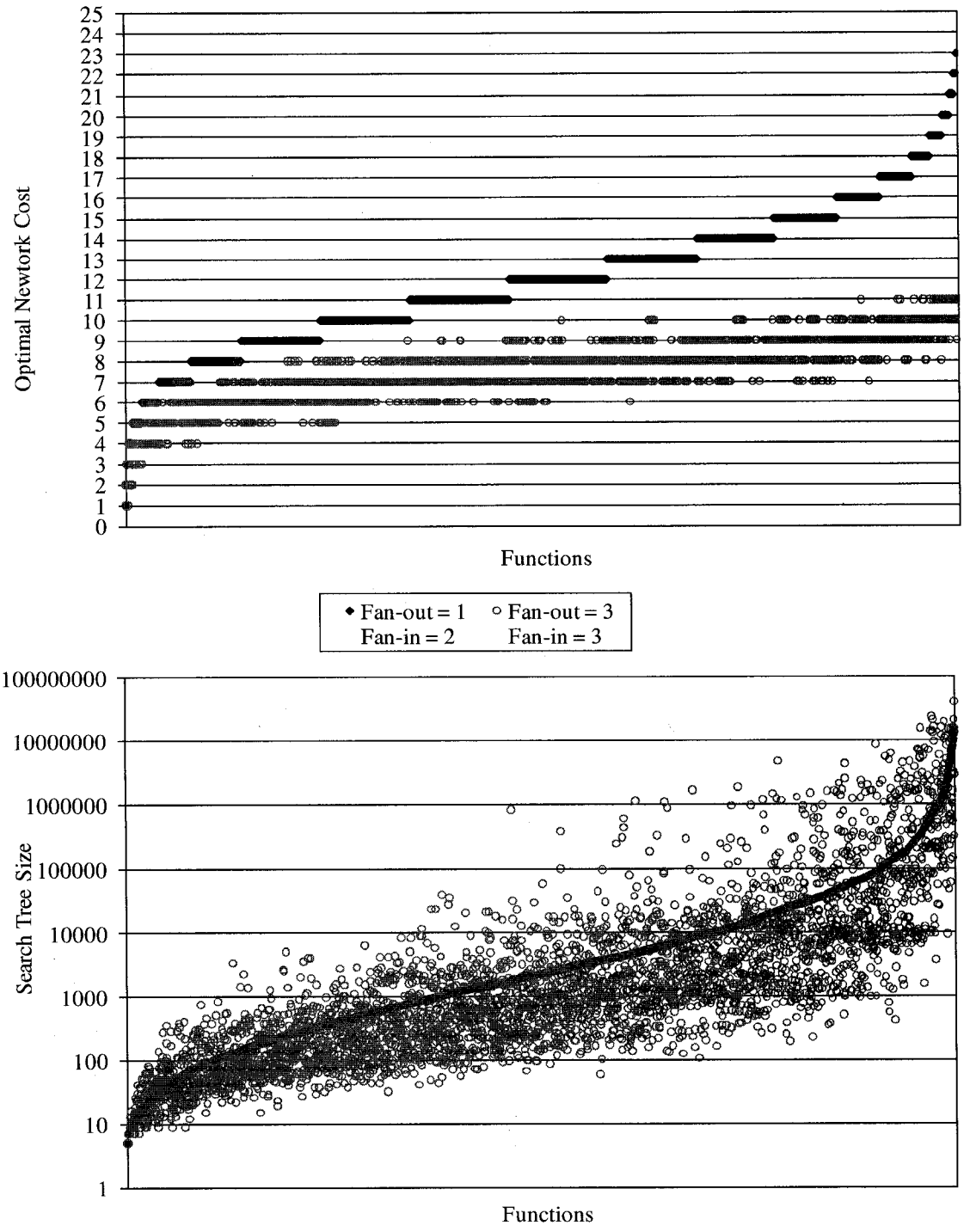


Figure 5.5: Graphs of Results when Fan-in = 3 and Fan-out = 3

By increasing the fan-in and fan-out restrictions in this variation compared to the previous variation, the cost of the optimal networks will decrease since more connections between the gates are allowed. The increase in the number of possible connections results in an increase in the size of the connectible sets. However, this results in an increase of the branch-width by an average of 19% to approximately 2.45. Fewer implications occur in the search as a result of the fan-in increase. This can cause the size of the search tree to increase for some functions. However, a decrease in the optimal cost will cause the size of the search tree to be reduced for other functions. Overall, a majority of the functions see a decrease in the size of the search tree.

5.2.3 Removal of Fan-in and Fan-out Restrictions

The final variation on the algorithm with regards to the fan-in and fan-out restrictions is the removal of both restrictions. The gate nodes in these networks can have any size fan-out sets as in the original networks, however now the fan-in sets can be any size as well. By removing the fan-in and fan-out restrictions there is greater freedom in the connections that can be made within the network.

5.2.3.1 Changes to the algorithm

Once again the `UpdateConnectibleSet` procedure must be changed to allow any valid connectible node into a connectible set regardless of the size of the fan-in or fan-out sets.

The `PropagateFunctionalImplications` procedure will need to be changed as well. Now that the fan-in limit has been removed, the function of a gate node $(i, j_0, j_1, \ldots, j_k)$ is $f_i = f'_{j_0} \vee f'_{j_1} \vee \cdots \vee f'_{j_k}$. Since a new node can always be added to the fan-in set of a gate node, a pseudo-node 0, with function $f_0 = [0,1]$ must always be considered as an input to the gate node when functional implications are performed. Thus the relation between a node's input and output function is expressed as follows:

$$\mathrm{ON}_i = \mathrm{OFF}_{j_0} \vee \cdots \vee \mathrm{OFF}_{j_k} \vee 0$$
$$\mathrm{OFF}_i = \mathrm{ON}_{j_0} \wedge \cdots \wedge \mathrm{ON}_{j_k} \wedge 0$$

Therefore functional implications can only be made to the on-set of the global functions. These implications are:

- Forward implications:

  $\mathrm{OFF}_{j_0} \rightarrow \mathrm{ON}_i$, ... , $\mathrm{OFF}_{j_k} \rightarrow \mathrm{ON}_i$

- Backward implications:

  $\mathrm{OFF}_i \rightarrow \mathrm{ON}_{j_0}$, ... , $\mathrm{OFF}_i \rightarrow \mathrm{ON}_{j_k}$

The global functional implications must be removed as well. Both types of global functional implications were based on the assumption that one of the two fan-in nodes at the point of the reconvergent fan-out must cover the on-set minterm from this origin node. This assumption can no longer be made with

this variation since a new node can always be added to the fan-in set of a gate node to cover a minterm. Thus, global functional implications of this type can no longer be performed.

All other pruning techniques and structural implications can remain unchanged since the properties of the network that allowed for these techniques to be used have remained unchanged.

5.2.3.2 Experimental Results

Two tables of results using this variation are given in Figure 5.6. The fan-in and fan-out of the gate nodes are unrestricted. Table (a) gives a summary of the results obtained with this variation on the set of representative functions for the P-equivalence classes of 2, 3, and 4 inputs. Table (b) gives a summary of results obtained with this variation on the classes of functions. A comparison of these results with the results from the original version is given in the graphs of Figure 5.7. The first graph compares the cost of the optimal network for each function while the second graph compares the size of the search tree for each function.

Inputs	Functions	Complete	Cost	Search Tree Size	Avg. Height	Avg. Width
2	8	8	24	54	3.15	2.00
3	68	68	327	1,904	5.68	2.09
4	3,904	3,904	27,861	12,401,238	11.13	2.38

(a) Representative Functions

Function Class	Max Input Completed	Cost	Search Tree Size	Avg. Height	Avg. Width
AND	9	16	96	4.571	2.000
NAND	9	8	96	4.571	2.000
OR	8	42	7,349	9.506	2.223
NOR	9	49	7,349	9.506	2.223
XOR	4	21	167,763	12.760	2.344
XNOR	4	22	135,983	15.617	2.367
MAJ	5	21	5,867,720	10.863	2.188
MUX	6	34	285	7.184	2.347

(b) Classes of Functions

Figure 5.6: Results of Fan-in and Fan-out Restriction Removal

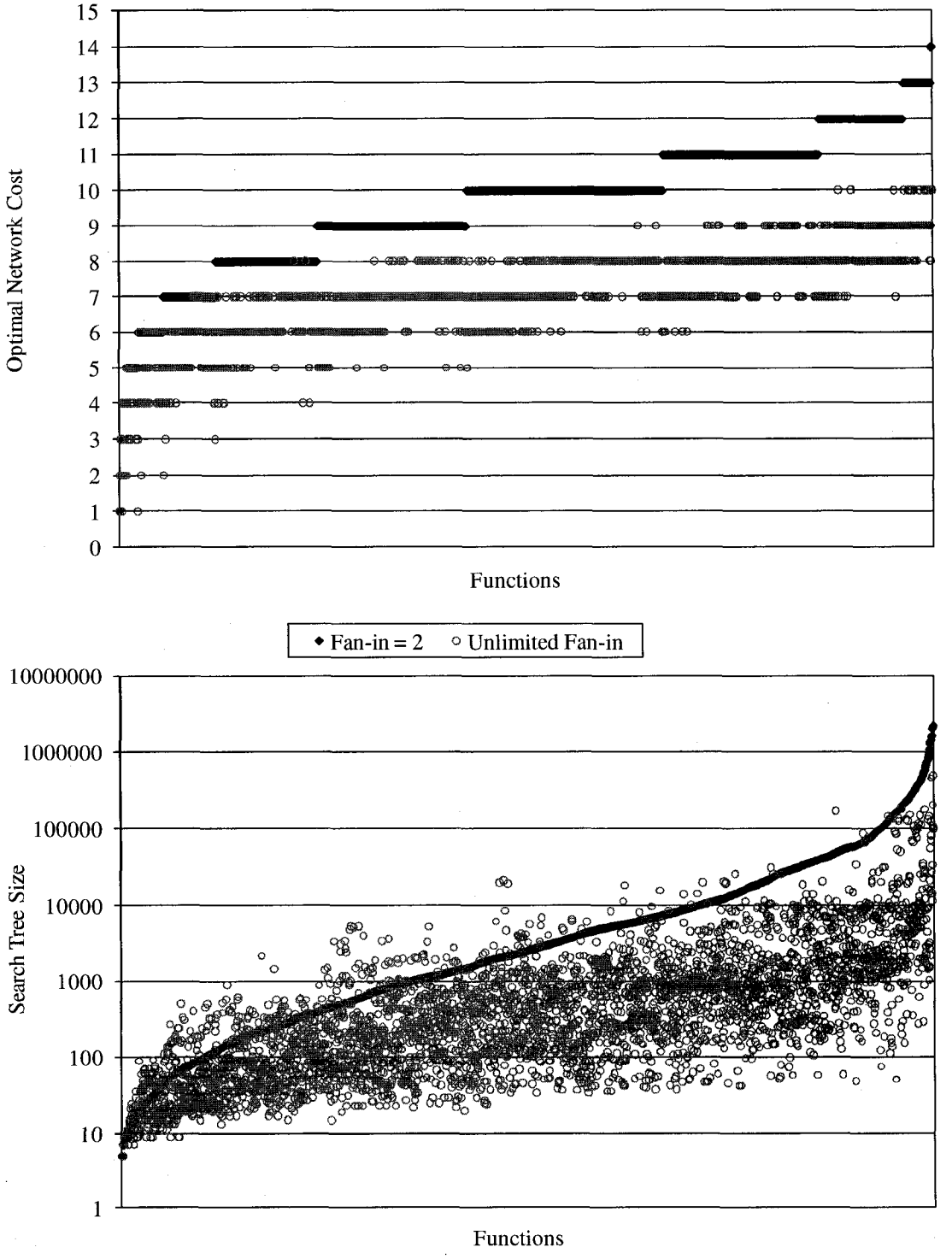


Figure 5.7: Graphs of Results when Fan-in and Fan-out Restrictions are Removed

The removal of the fan-in limit on the gate nodes has several effects on the search. The first is on the cost of the optimal network. By allowing a larger fan-in set for the gate nodes, more nodes can contribute in the covering of the on-set minterms from the global function of a gate node, and thus fewer nodes will be required to complete a network. The cost of an optimal network from this variation is always less than or equal to the cost of the optimal network found with the original version. The results show that 95% of the functions saw a reduction in the optimal cost for a network. The average optimal cost was reduced by approximately 25% or 2.5 gates. Of the 3,980 function from the P-equivalence classes, 3,907 had optimal networks containing gates with fan-in larger than two. The maximum fan-in of a gate was six. Twenty-two of the networks had a gate with six inputs.

The second effect that results from allowing unlimited fan-in is that fewer functional implications are possible. Without the fan-in bound, only a single level of functional implications is possible for any change in the global function of a node. This can reduce the time spent updating the network after a covering is made. It can also cause more coverings to be necessary since fewer coverings will be performed simply as a result of the functional implications. The result would be an increase in the height of the search tree. However, such an increase is not seen. The reduction in the number of gates in the network counteracts any increase in the number of coverings required due to fewer functions implications.

The cost of the network plays the largest role in determining the size of the search tree here. The significant decrease in the size of the optimal networks results in an equally significant decrease in the size of the search trees. An average 25% decrease in the cost of the networks translates into an average 86% reduction in the size of the search tree.

Since the optimal networks tend to be smaller compared to the original results, optimal networks for more of the functions from the classes of functions are able to be completed within the time limit. However on closer inspection one can discover that the algorithm was unable to find and prove optimal networks for functions that require more than 16 gates. This is similar to the limitations discovered on the original version of the algorithm in Chapter 4.

5.3 Level Restriction

A reduction in the size of the search space can be achieved by imposing a level restriction on the network. This allows the algorithm to complete the search on functions requiring larger cost networks than what has been completed previously. A level restriction will be added to two versions of the algorithm. The first version will be based on the original version where gate nodes have a fan-in restriction of 2. With both a level and fan-in restriction, networks for some functions may no longer exist under these constraints. An evaluation of which functions can be completed and which cannot be completed will be given. The second version will then remove the fan-in restriction so that only a level restriction is placed on the networks. This will allow optimal networks for all functions to be found within the restriction. A comparison of these results to those obtained in Section 5.2.3 can then be completed.

5.3.1 Restrictions: Level = 3, Fan-in =2

The algorithm is able to enforce this level restriction with a simple change in the `UpdateConnectibleSet` procedure. In addition to the functional and structural constraints that a node must fulfill in order to be included in a connectible set, a level constraint must also be added. This level constraint specifies that the addition of the node l as an input to a gate node i will not create a network with more than 3 levels. Aside from this additional constraint no other changes need to be made to the original version.

The purpose of adding this level restriction was to reduce the size of the search space. However, because of this level restriction a rise in the cost of an optimal network should be expected as well.

Results of this version are given in Figure 5.8. As expected, networks for only a very few number of functions could be found because of the restrictions placed on the networks. The details of the search in the tables are given for only those functions on which the search was completed and a network found. The largest network that was completed contained 7 gates. There were 7 such functions, one with 3 inputs and the remaining with 4 inputs. As the tables indicate, when the number of inputs to the network increase, fewer networks are to be created because of the restrictions. Only 1% of the 4-input functions have networks, while all 2-input functions have a network. Because the networks that were found are so small, it is hard to perform a real comparison to detect how this restriction effects the search space. Thus the fan-in restriction must be removed.

Inputs	Functions	Complete	Cost	Search Tree Size	Avg. Height	Avg. Width
2	8	8	24	64	3.26	2.03
3	68	23	97	909	3.89	2.09
4	3904	45	233	42,260	3.10	2.07

(a) Representative Functions

Function Class	Max Input Completed	Cost	Search Tree Size	Avg. Height	Avg. Width
AND	2	2	5	2.67	2.00
NAND	4	9	29	3.60	2.00
OR	2	3	5	2.67	2.00
NOR	2	4	5	2.67	2.00
XOR	2	4	15	5.88	2.00
XNOR	2	5	29	5.18	2.00
MAJ	2	2	5	2.67	2.00
MUX	3	4	10	4.00	2.25

(b) Classes of Functions

Figure 5.8: Results of Level = 3 Restriction

5.3.2 Restriction: Level = 3, Unrestricted Fan-in

The two tables of results using this variation of the algorithm are given in Figure 5.9. Table (a) shows the result of this variation on the set of representative functions for the P-equivalence classes on 2, 3, and 4 inputs. Table (b) shows the results of this variation on the classes of functions.

Now that the fan-in restriction has been removed, networks for all functions are completed. Effects of this level restriction can be discovered when they are compared to the results given in Figure 5.6. A comparison of these two versions in terms of the optimal network cost and search tree size for individual functions are shown in Figure 5.10. The first graph in Figure 5.10 shows that the search tree size is reduced for most functions when the fan-in limit is imposed. On average the search tree was reduced by 41% when the level constraint was added due to the reduction in the solution space. The overall reduction in the search tree size results in a 22% reduction in the amount of time required to complete the search. While the search space was reduced due to a reduction in the overall size of the solution space, the cost of the optimal networks increased as shown in the second graph in Figure 5.10. For an average function, the cost increased by approximately 8%.

The details of the search tree changed as expected. The average width of a branch in the search tree increased over the previous results with no level restriction. This increase is due to the necessary increase in the size of the fan-in of a gate node when a level restriction is imposed. The search tree height also increased due to an increase in the number of gates in the optimal network that are necessary to maintain the level restriction.

Inputs	Functions	Complete	Cost	Search Tree Size	Avg. Height	Avg. Width
2	8	8	24	56	3.14	2.04
3	68	68	334	1,620	5.88	2.19
4	3904	3904	30,344	1,637,156	12.11	2.45

(a) Representative Functions

Function Class	Max Input Completed	Cost	Search Tree Size	Avg. Height	Avg. Width
AND	9	16	96	4.571	2.000
NAND	9	8	96	4.571	2.000
OR	9	52	138	4.601	2.000
NOR	9	60	96	4.571	2.000
XOR	4	25	48,867	15.293	2.603
XNOR	4	26	9,780	13.874	2.556
MAJ	9	257	13,999	7.875	2.012
MUX	9	33	2,769	2.380	2.380

(b) Classes of Functions

Figure 5.9: Results of Level Restriction = 3 and No Fan-in Restriction

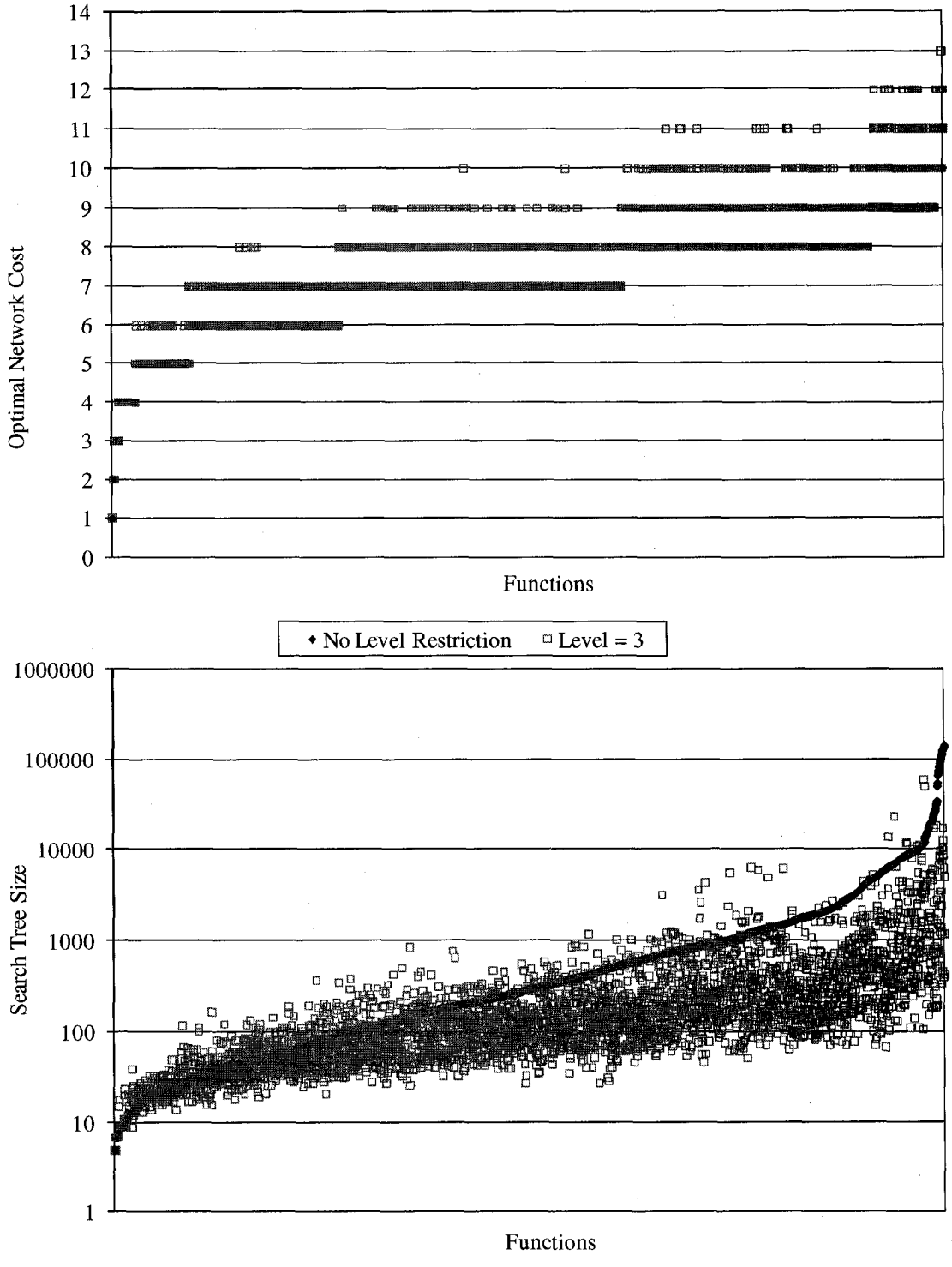


Figure 5.10: Graphs of Results of Level Restriction = 3 and No Fan-in Restriction

This level restriction produces an increase in the number of gates for an optimal network, while a decrease in the size of the search tree is seen. Because of the reduction in the search space larger functions from some of the function classes were completed within the time limit.

5.4 Alternate Cost Functions

The cost function used in the original version of the algorithm is based solely on the number of gates in the network. Alternate properties of the network can be used as part of the cost function however. In this section results of the algorithm with several different cost functions are provided. These demonstrate the types of cost functions available as well as the effects that these cost functions have on the algorithm and the optimal networks.

5.4.1 Cost Function using Gates and Interconnect

The first cost function that will be employed ranks networks according to the number of gates in the network as well as the number of interconnections that exist between nodes in the networks. These two properties of the network can be combined into a cost function as follows: $\text{Cost} = 10(\text{gates}) + \text{edges}$. This cost function prioritizes the number of gates as the first objective while including the number of edges as a second objective. In some cases an increase in the number of gates may be traded for a steep reduction in the number of interconnections.

5.4.1.1 Changes to the Algorithm

The only change that must be made in the algorithm is the computation of the cost function. In the original version, the value of the cost function for a given network is maintained by the network data structure. When a new gate is added to the network the cost is increased. The same process can be performed for this variation, but now the cost function must be updated any time a gate or edge is added to the network.

5.4.1.2 Experimental

Because additional constraints are added to the cost function, the size of the search tree should increase. More of the solution space must be explored to guarantee the optimality of the network with regards to both the number of gates and the number of interconnections.

The tables in Figure 5.11 give the results of the algorithm using this new cost function. In table (a), the fourth column give the sum of the costs of the optimal networks for all functions in the class; the fifth and sixth columns give the specific details of this cost dividing it into the sum of the gates and the sum of the edges of the optimal network. Similar details for the cost function are given in columns three through five in table (b). The remainder of each table gives the details of the search performed by the algorithm. The

graphs in Figure 5.12 compare the number of gates in the optimal circuit and the size of the search tree for individual functions when this variation and the original version are used.

Inputs	Functions	Complete	Cost	Gates	Edges	Search Tree	Avg. Height	Avg. Width
2	8	8	277	24	37	70	3.27	2.02
3	68	68	4,730	405	680	33,441	7.89	2.25
4	3,904	3,904	448,278	37,936	68,918	1,797,451,468	15.81	2.49

(a) Representative Functions

Function Class	Max Input Completed	Cost	Gates	Edges	Search Tree	Avg. Height	Avg. Width
AND	7	483	42	63	7,918,363	15.56	2.27
NAND	7	417	36	57	7,918,363	15.56	2.27
OR	6	510	45	60	4,448,834	11.21	2.39
NOR	6	565	50	65	4,448,834	11.21	2.39
XOR	4	288	24	48	948,844	18.77	2.51
XNOR	4	318	27	48	2,199,586	19.48	2.74
MAJ	4	188	16	28	4,539	8.66	2.24
MUX	6	591	50	91	531,457	13.66	2.48

(b) Classes of Functions

Figure 5.11: Results with Cost Function: $\text{Cost} = 10(\text{gates}) + \text{edges}$

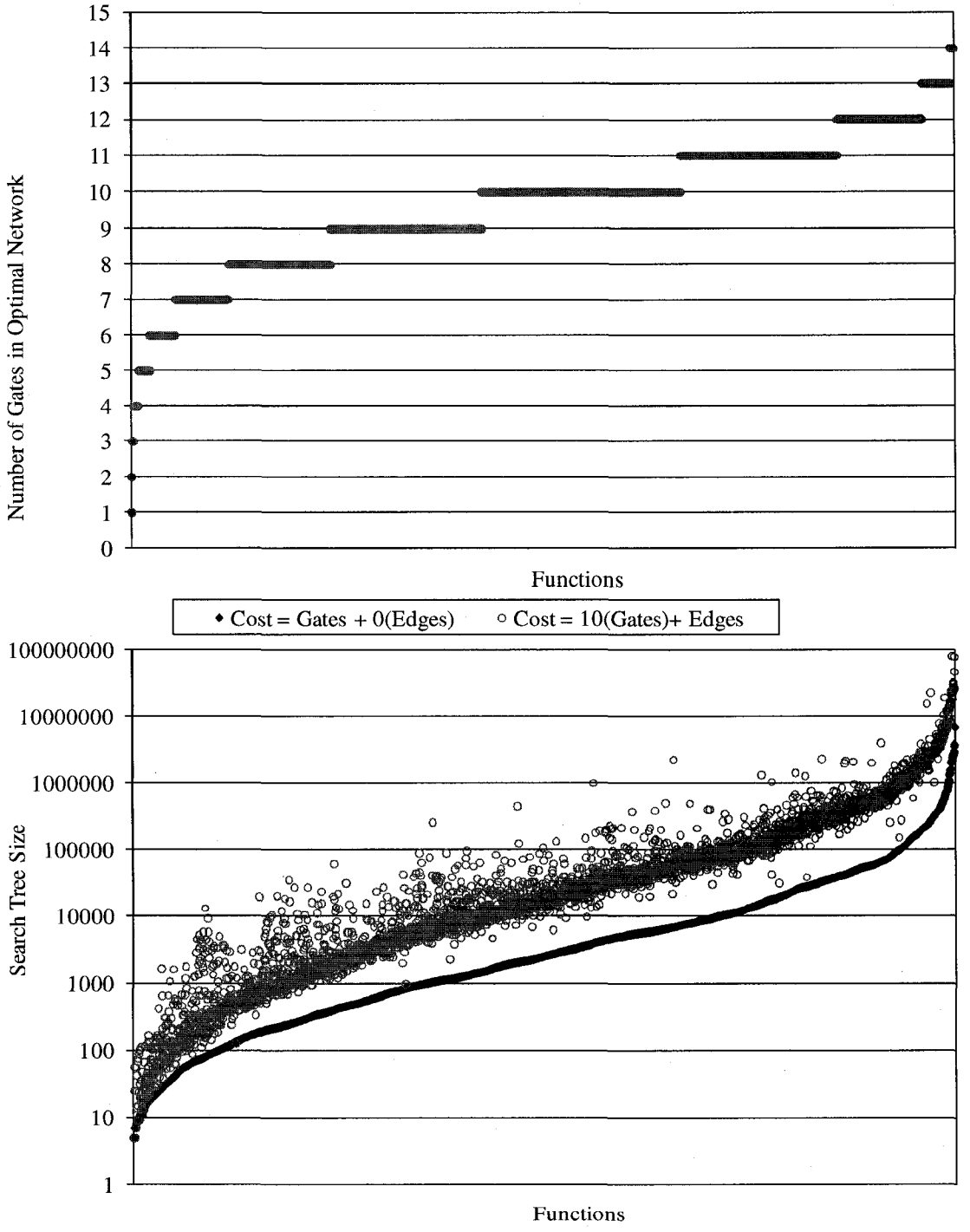


Figure 5.12: Graphs of Results with Gate and Edges Cost Function

Due to the weight placed on the number of gates in this cost function, no trade-off was made to reduce the number of interconnections by increasing the number of gates in the optimal network of a function. The cost function only ranked networks with the same number of gates based on the number of edges in the network. Of the 3,980 functions, 1,937 of them had networks with fewer edges than the one found by the original version.

Since the cost function is based on both the number of gates and number of edges in the network, more of the solution space must be searched in order to guarantee the optimality of the network under both constraints in the cost function. On the P-equivalence class functions, the search tree produced by the algorithm increased by an average of 858%.

5.4.2 Cost Function using Gates and Levels

An alternate version of the cost function uses both the number of gates and the number of levels to rank networks. Once again, these two properties can be combined into a single cost function: $\text{Cost} = 2(\text{gates}) + \text{levels}$. With this cost function, the number of gates is prioritized as the first objective although not as strongly as in the previous cost function variation, while the number of levels in the networks is the second priority. With this cost function, more cases are expected where an increase in the number of gates is traded for a reduction in the number of levels.

5.4.2.1 Experimental Results

The tables in Figure 5.13 give the results of the algorithm using this new cost functions. In table (a) of Figure 5.13 the fourth column gives the sum of the costs of the optimal networks for all functions in the class; the fifth and sixth columns give the specific details of this cost dividing it into the sum of the gates and the sum of the levels of the optimal networks. Similar details for the cost function are given in columns three through five in table (b). The remainder of each table gives the details of the search performed by the algorithm including the time for the search to complete, the size of the search trees produced by the algorithm and the average width and height for these search trees. The graphs in Figure 5.14 compare the number of gates in the optimal circuit and the size of the search tree for individual functions when this variation and the original version are used.

Inputs	Functions	Complete	Cost	Gates	Levels	Search Tree Size	Avg. Height	Avg. Width
2	8	8	67	24	19	64	3.28	2.03
3	68	68	1,081	405	271	6,532	6.84	2.20
4	3904	3904	97,662	38,104	21,454	167,541,866	13.59	2.49

(a) Representative Functions

Function Class	Max Input Completed	Cost	Gates	Levels	Search Tree Size	Avg. Height	Avg. Width
AND	7	112	42	28	307,976	11.82	2.15
NAND	7	94	36	22	307,976	11.82	2.15
OR	6	112	45	22	169,322	9.28	2.29
NOR	6	127	50	27	169,322	9.28	2.29
XOR	4	63	24	15	72,397	16.13	2.54
XNOR	4	69	27	15	105,756	16.23	2.66
MAJ	4	42	16	10	277	6.12	2.16
MUX	6	127	50	27	65,556	11.00	2.44

(b) Classes of Functions

Figure 5.13: Results with Cost Function: $\text{Cost} = 2(\text{gates}) + \text{levels}$

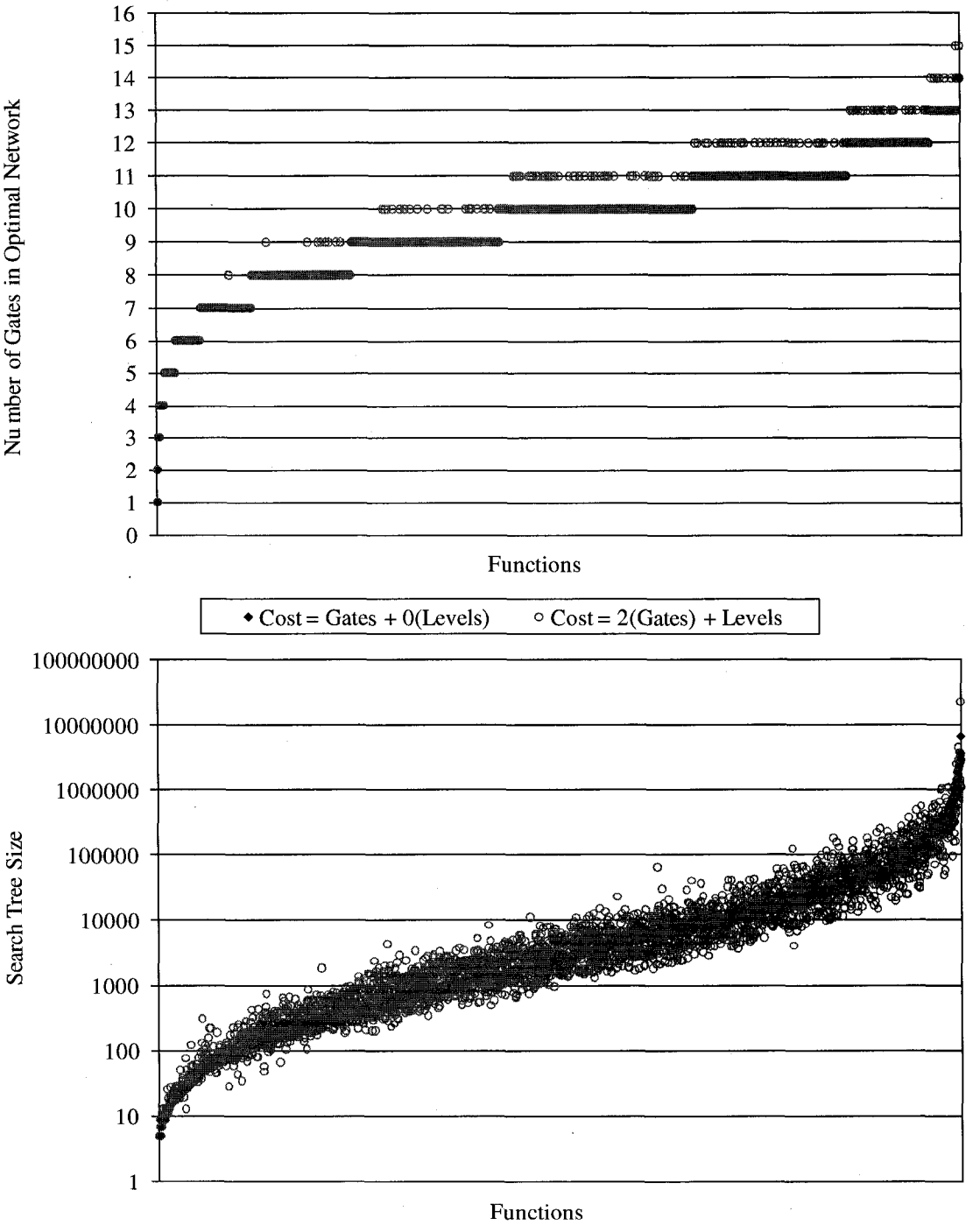


Figure 5.14: Graphs of Results with Gate and Levels Cost Function

The results show that there are some cases where the number of gates in the network is increased to obtain a decrease in the number of levels, producing a lower cost network. This only happens however on a few of the larger functions. 166 of the 3,980 functions have more gates in the optimal network found here than the one found with the original version. The largest network from this group uses 15 gates rather than 14 previously.

The majority of functions saw a slight increase in the size of the search tree. This is due to an increase in the amount of the search space that must be considered. The increase was much smaller than the increase observed with the previous cost function; however, it was large enough to prohibit four networks from the classes of functions from being completed in the allotted time. These four were the 6-input OR, the 6-input NOR, the 5-input MAJORITY, and the 6-input MULTIPLEXER. In each of these four cases this version of the algorithm found a network with the same number of gates as the basic version of the algorithm but since a larger portion of the solution space must be searched to determine if the network is optimal with respect to both the number of gates and the number of levels, the search was not able to be completed in time.

5.5 Alternate Building Blocks

We chose to use the building block set {NAND2} for our original version for several reasons including the simplicity of the set as well as the prevalence of this gate in logic synthesis. However, the algorithm can be changed to perform synthesis with any complete set of logic gates. This will require a reworking of the covering conditions and the functional implications in the algorithm. In this section the details on how this can be done and results of the algorithm using the complete sets {NOR2} and {AND2, OR2, NOT} are given.

5.5.1 Building Block Set {NOR2}

5.5.1.1 Changes to the Algorithm

The duality of the Boolean functions NAND and NOR make the conversion of the algorithm using NAND gates to one that uses NOR gates relatively simple. In Chapter 2 the relationship among a node's input and output functions was given for the NAND2 gate. This same relationship for a NOR2 gate is:

$$ON_i = OFF_j \wedge OFF_k$$
$$OFF_i = ON_j \vee ON_k$$

Maintaining the consistency of this relationship requires the following functional implications:

- Forward implications:

  $ON_j \rightarrow OFF_i$ and $ON_k \rightarrow OFF_i$

  $OFF_j \wedge OFF_k \rightarrow ON_i$

- Backward implications:

$ON_i \rightarrow OFF_j$ and $ON_i \rightarrow OFF_k$

$OFF_i \wedge OFF_j \rightarrow ON_k$ and $OFF_i \wedge OFF_k \rightarrow ON_j$

This relationship was also used to determine the set of functionally consistent nodes as described in Section 2.5. This is the set of nodes l that that can be connected as an input to node i without violating the function constraints of the node. The consistency constraint for a NOR2 gate can be stated as follows:

$ON_l \wedge ON_i = 0$

and

$OFF_l \wedge OFF_k \wedge OFF_i = 0$

The final property dependent on the local function of a gate is the covering property. A minterm m in the off-set of a NOR gate (i, j, k) is covered if $m \leq ON_j \vee ON_k$, i.e., m is contained in the on-set of either input. This implies that the off-set of the NOR gate nodes will be divided into two sets: the covered and the uncovered sets.

5.5.1.2 Experimental Results

By employing these functional changes into the procedures of the algorithm a new version is created which provides the optimal network using NOR2 gates. The results of this version are presented in Figure 5.15 and Figure 5.16.

Inputs	Functions	Complete	Cost	Search Tree	Avg. Height	Avg. Width
2	8	8	24	58	3.25	2.00
3	68	68	405	5,971	6.72	2.21
4	3904	3904	37936	161,779,078	12.93	2.41

(a) Representative Functions

Function Class	Max Input Completed	Cost	Search Tree	Avg. Height	Avg. Width
AND	6	45	2,534,439	10.37	2.25
NAND	6	50	2,534,439	10.37	2.25
OR	7	42	1,415,828	12.90	2.10
NOR	7	36	1,415,828	12.90	2.10
XOR	4	26	167,927	13.51	2.40
XNOR	4	25	48,376	16.36	2.48
MAJ	4	20	32,707	8.77	2.19
MUX	6	52	54,843	11.33	2.40

(b) Classes of Functions

Figure 5.15: Results with Building Block {NOR2}

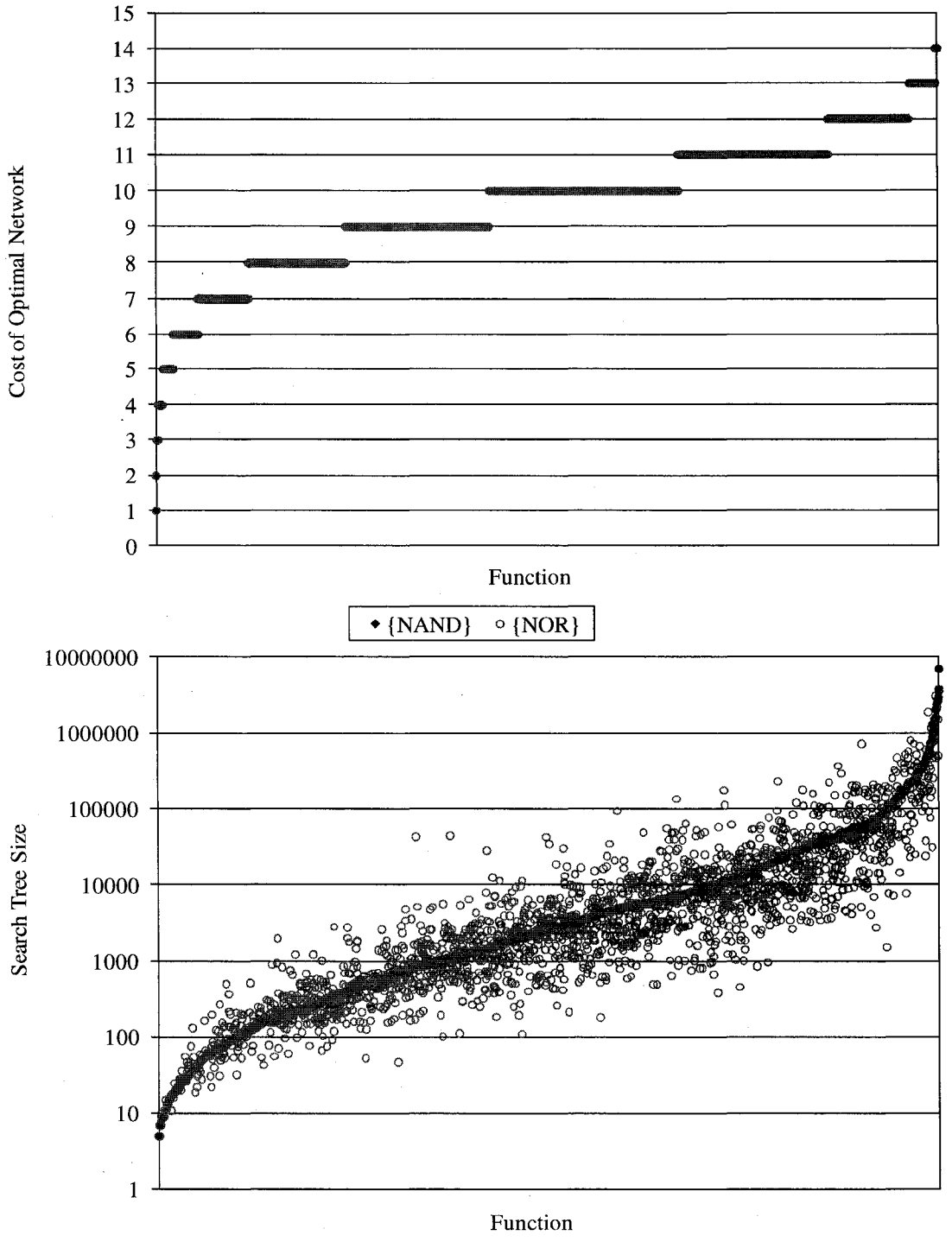


Figure 5.16: Graphs of Results with Building Block {NOR2}

As expected, the duality of the NAND and NOR functions causes the algorithm to produce similar results when the NOR gate is used as a building block. The graphs in Figure 5.16 show a comparison of the cost of an optimal network and the size of the search tree for each function using the original version compared with the data of its dual function using NOR gates. Thus, the costs of the two networks are the same. The second graph shows that this version of the algorithm searches the space in the same way producing search trees which are similar to those produced when NAND gates are used.

5.5.2 Building Block Set {AND2, OR2, NOT}

The algorithm can also be converted to find optimal networks using the building blocks AND2, OR2, and NOT. This will require some additional work since more than one gate type exists in this building block set. First the relationship among a node's input and output functions must be determined for each gate type. Then from this the functional implications, functional connectibility constraint and covering definition for each gate type is generated.

5.5.2.1 Algorithmic Changes

The relationship among a node's input and output functions for gate type AND2, OR2, and NOT can be expressed as follows:

The AND gate (i, j, k): $ON_i = ON_j \wedge ON_k$
$OFF_i = OFF_j \vee OFF_k$

The OR gate (i, j, k): $ON_i = ON_j \vee ON_k$
$OFF_i = OFF_j \wedge OFF_k$

The NOT gate (i, j): $ON_i = OFF_j$
$OFF_i = ON_j$

The consistency of these relationships is maintained using the following implication rules:

AND gate

- Forward implications: $OFF_j \rightarrow OFF_i$ and $OFF_k \rightarrow OFF_i$
  $ON_j \wedge ON_k \rightarrow ON_i$
- Backward implications: $ON_i \rightarrow ON_j$ and $ON_i \rightarrow ON_k$
  $OFF_i \wedge ON_j \rightarrow ON_k$ and $OFF_i \wedge ON_k \rightarrow ON_j$

OR gate

- Forward implications: $ON_j \rightarrow ON_i$ and $ON_k \rightarrow ON_i$
  $OFF_j \wedge OFF_k \rightarrow OFF_i$
- Backward implications: $OFF_i \rightarrow OFF_j$ and $OFF_i \rightarrow OFF_k$
  $ON_i \wedge OFF_j \rightarrow OFF_k$ and $ON_i \wedge OFF_k \rightarrow OFF_j$

NOT gate

- Forward implications: $OFF_j \rightarrow ON_i$
  $ON_j \rightarrow OFF_i$
- Backward implications: $OFF_i \rightarrow ON_j$
  $ON_i \rightarrow OFF_j$

Finally, the functional consistency constraint determines the set of nodes l that can be connected as an input of the node i while maintaining the functional relationship of the gate. The consistency constraints for each gate type can be stated as follows:

AND gate: $OFF_l \wedge ON_i = 0$ and $ON_l \wedge OFF_k \wedge ON_i = 0$

OR gate: $ON_l \wedge OFF_i = 0$ and $OFF_l \wedge ON_k \wedge OFF_i = 0$

NOT gate: $OFF_l \wedge OFF_i = 0$ and $ON_l \wedge ON_i = 0$

Finally, the gate type determines the covering property that drives the search. For an AND gate, the minterms in the **off-set** of a node (i, j, k) are covered if the minterms appear in the **off-set** of either j or k. The minterms in the **on-set** of an OR gate (i, j, k) are covered if the minterms appear in the **on-set** of either j or k. For the NOT gate, the minterms in both the on- and off-set must be covered by the gate's only fan-in. Thus either the on-set or off-set is sufficient for use in this covering property. When these additions are made in the appropriate procedures of the algorithm, the gate type will be used to determine which function properties should be followed.

The use of multiple gate types in the network adds to the complexity of adding a new gate to the network. When a new gate is added, a choice of which type of gate it will be is necessary. Thus a single branch in the network becomes three branches. This implies that structural implications are no longer possible in this case. A structural implication in the original version occurred when the only option for covering a minterm was by adding a new gate to the network. When only one gate type existed, there was only one possible way to perform this covering so an implication rather than a branch resulted. However, with this set of building blocks, when the only option for covering a minterm is to add a new gate to the network, a choice remains as to which type of gate should be used. Thus a branching step rather than an implication is needed. This leads to a removal of the structural implication procedure from the algorithm for this version and an additional rule to the minterm selection. The minterm selection heuristic will now select a node first if the only option is to add a new gate to the network. If no such node exists, then the heuristic will proceed as before.

5.5.2.2 Experimental Results

The results of the algorithm using the building block set {AND2, OR2, NOT} are given in Figure 5.17 and Figure 5.18. Table (a) from Figure 5.17 shows the result of this variation on the set of representative functions for the P-equivalence classes on 2, 3, and 4 inputs, while table (b) shows the results of this variation on eight of the classes of functions. The graphs in Figure 5.18 compare this version with the

original version. The cost of the optimal network as well as the size of the search tree is compared for individual functions.

Inputs	Functions	Complete	Cost	Search Tree	Avg. Height	Avg. Width
2	8	8	18	572	3.90	3.38
3	68	68	329	390,515	7.21	3.54
4	3,904	3,120	26,195	13,815,224,078	15.01	3.85

(a) Representative Functions

Function Class	Max Input Completed	Cost	Search Tree	Avg. Height	Avg. Width
AND	6	15	3,060	2.60	2.20
NAND	6	20	4,979	3.20	2.20
OR	6	15	2,805	2.60	2.60
NOR	6	20	6,216	3.20	2.20
XOR	3	12	17,632	5.00	2.00
XNOR	3	12	10,852	5.00	2.00
MAJ	4	12	101,913	3.33	2.33
MUX	3	4	93	3.00	3.00

(b) Classes of Functions

Figure 5.17: Results with Building Blocks {AND2, OR2, NOT}

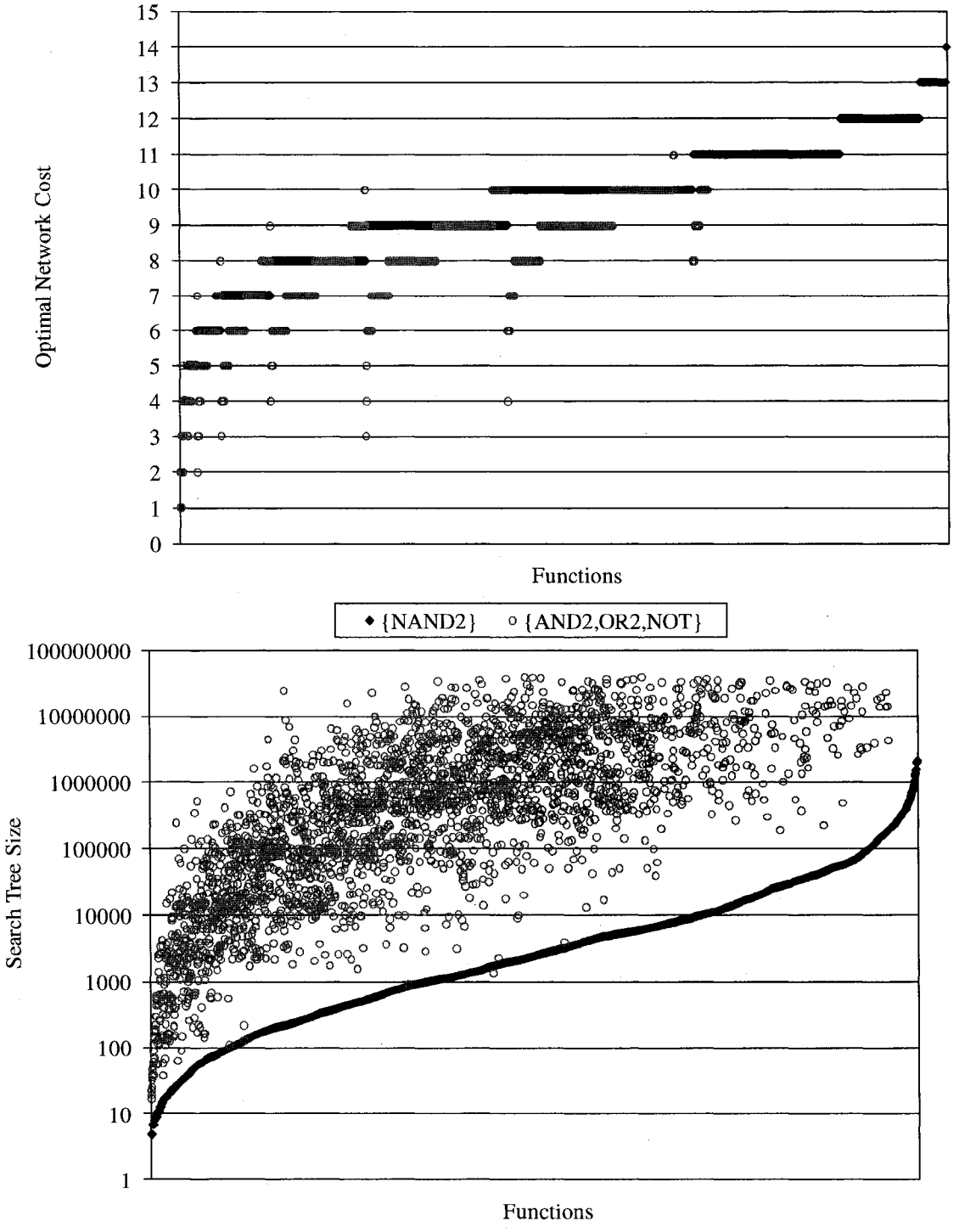


Figure 5.18: Graph of Results with Building Blocks {AND2, OR2, NOT}

Significant changes to both the cost of the optimal networks and the size of the search trees result when this new set of building blocks is used. One change is that the cost of the optimal network decreases for a majority of the functions. This is to be expected since more gates are available to construct the networks. On average, the cost of the network was reduced by 1 gate.

The increase in the number of building blocks available also creates significant changes in the size of the search tree. Every time a new gate is added to the networks, the choice of which type of gate to use must be considered. This increases the width of the branches by an average of 1.5. This will also cause the height of the paths in the search tree to increase since structural implications are no longer possible. The average path height increases from 12 nodes to 15. These changes to the properties of the search tree result in the dramatic increase in the size of the search trees seen in the second graph t in Figure 5.18.

5.5.3 General Building Block Set

Any complete set of building blocks can be employed in this synthesis algorithm. The logic operations described here are vertex functions [Karp 61]. Each function evaluates to one of its values (1 or 0) for exactly one assignment of the input variables and the opposites values for all others. These types of functions lend themselves to the covering process very easily. Let $f(x, y)$ be a vertex function which evaluates to $b \in \{0,1\}$ on the input assignment a and evaluates to $\{0,1\} \setminus b$ for every other assignment. If f is used as a building block, then the relationship among the node's input and output functions is based on the function f. The functional implications and functional consistency constraint will then follow from this relationship. The covering property for this gate is determined by the value b. If $b = 0$ then the off-set of gate node must be covered by the input nodes. If $b = 1$ then the on-set of the gate node must be covered by the input nodes.

More complex functions can also be used as building blocks. Once again the relationship among the node's input and output functions, the functional implications required to maintain this relationship, and the functional consistency constraint will all depend on the function used for this building block. When a non-vertex function is used as a building block, both the on- and off-sets of the gate must be covered by its inputs.

For example, the function relationship, forward implications, and covering rule for a an XOR gate node are as follows:

- The relations among the XOR gate node's input and output function:

$$ON_i = (ON_j \wedge OFF_k) \vee (OFF_j \wedge ON_k)$$

$$OFF_i = (ON_j \wedge ON_k) \vee (OFF_j \wedge OFF_k)$$

$$DC_i = DC_j \vee DC_k$$

- Forward implications for the XOR gate:

$$(ON_j \wedge OFF_k) \rightarrow ON_i$$
$$(OFF_j \wedge ON_k) \rightarrow ON_i$$
$$(ON_j \wedge ON_k) \rightarrow OFF_i$$
$$(OFF_j \wedge OFF_k) \rightarrow OFF_i$$

- The global functions at the inputs of the XOR gate must cover both on- and off-sets for the gate to be considered covered. Since the covering minterms are dependent on the relationship between global functions of the fan-in nodes, the covering must be performed by considering pairs of fan-in nodes.

5.6 Complemented Inputs

The final variation on BESS will be to allow complemented inputs as input nodes to the network. This will allow the network more options to use in the covering with no cost penalty. The result should be smaller networks and a smaller search tree. This change should allow the algorithm to find optimal solutions for a larger set of functions.

5.6.1.1 Changes to the Algorithm

The only addition that is necessary in order for the algorithm to complete this version is to add complemented input nodes to the initial network before the first SynthesizeNetwork procedure is called. For example, the initial network for the set of function $\{g_1 = x_1 \oplus x_2, g_2 = x_1 \wedge x_2\}$ will go from the network shown on the left in Figure 5.19 to the network shown on the right.

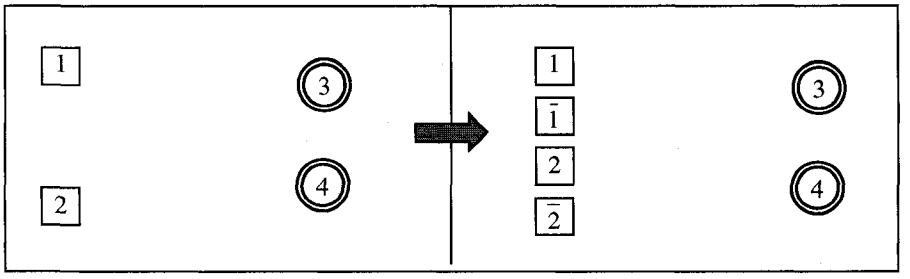


Figure 5.19: Initial Network with Complemented Inputs

5.6.1.2 Experimental Results

The results of the algorithm which allows for complemented inputs are given in Figure 5.20 and Figure 5.21. Figure 5.20 gives tables of results from this variation on the set of representative functions for the P-equivalence classes on 2, 3, and 4 inputs and the classes of functions. The graphs in Figure 5.21 compare the results of this variation to the original version on individual functions. The optimal cost and search tree size are compared.

Inputs	Functions	Complete	Cost	Search Tree	Avg. Height	Avg. Width
2	8	8	15	48	2.95	2.00
3	68	68	294	4,249	5.84	2.08
4	3,904	3,904	30,555	72,447,331	12.62	2.34

(a) Representative Functions

Function Class	Max Input Completed	Cost	Search Tree	Avg. Height	Avg. Width
AND	7	42	2,863,646	14.49	2.16
NAND	7	36	2,863,646	14.49	2.16
OR	7	36	255,603	11.64	2.08
NOR	7	42	255,603	11.64	2.08
XOR	4	20	120,344	14.64	2.30
XNOR	4	20	28,794	10.75	2.24
MAJ	5	25	258,362	10.40	2.19
MUX	6	44	22,546	12.12	2.34

(b) Classes of Functions

Figure 5.20: Results with Complemented Inputs

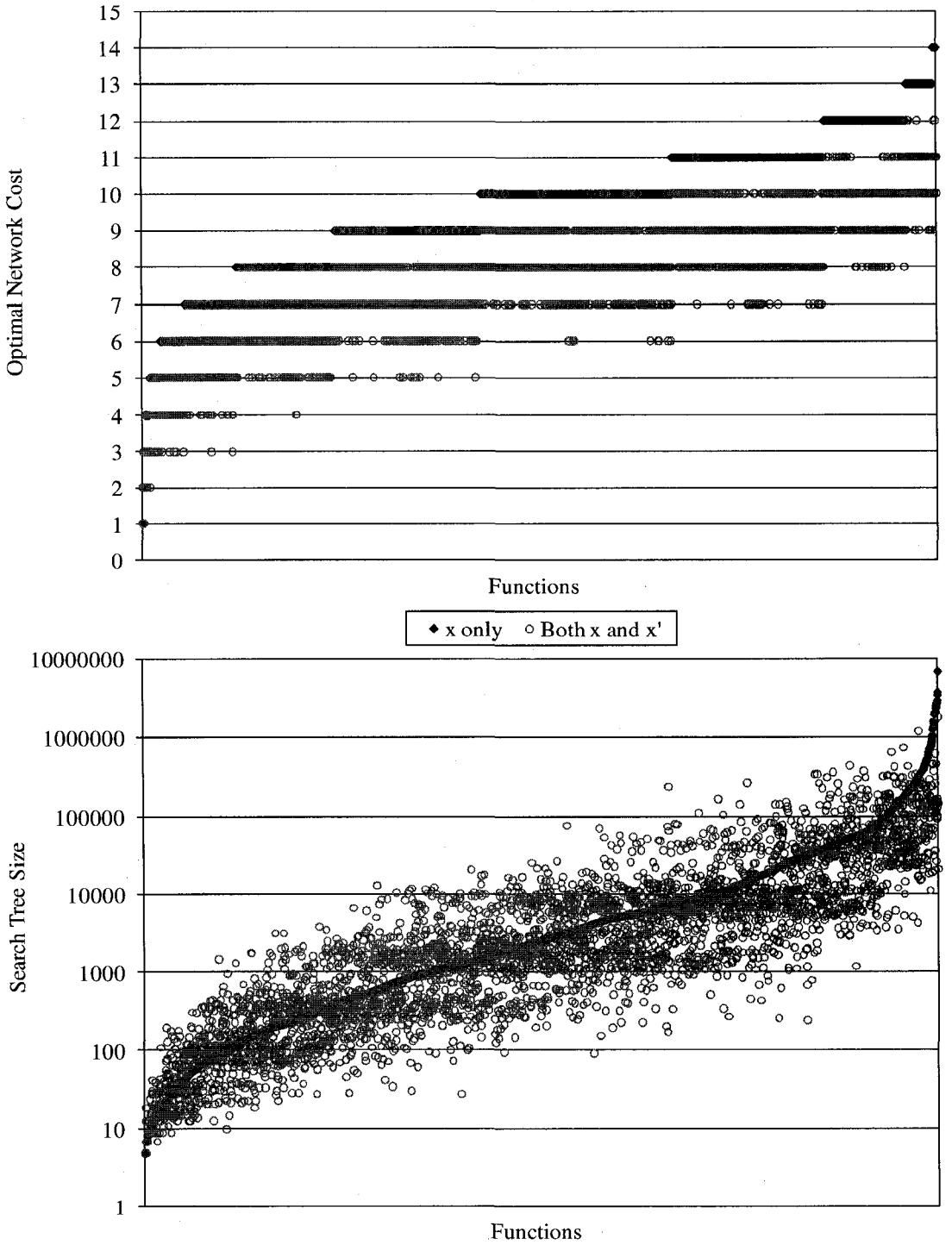


Figure 5.21: Graphs of Results with Complemented Inputs

The results show that the cost (number of gates) of the optimal networks is reduced for 96% of the functions. The average decrease is about 1.8 gates. This reduction in the cost helps to reduce the size and height of the search tree for the majority of functions as well. The size of the search tree is reduced by as much as 99% for some functions. The reductions are the result of the fact that separate nodes do not need to be created in order for the negation of the inputs to be used. This saves in the cost of the optimal network since fewer gate nodes are required to complete the network and it saves in the size of the search tree since fewer gate nodes require covering.

The average width of a branch in the search trees remains approximately the same despite the fact that twice as many input nodes are available to the network. This is due to the fact that it is not possible for both an input node and its complement to cover the same minterm. Therefore the branch width will not increase due to these additional complemented input nodes.

Similar reductions to the size of the optimal networks and the search required to complete these networks can be seen when the {AND2, OR2, NOT} building block set is used. In this case, however, the addition of the complemented inputs to the network allows us to eliminate the NOT gate from the building block set while still maintaining the completeness of the set. This allows further reductions in the search.

The results of the algorithm which allows complemented inputs and uses the building block set {AND2, OR2, NOT} are given in Figure 5.22 and Figure 5.23, while the same results using the building block set {AND2, OR2} are given in Figure 5.24 and Figure 5.25.

Inputs	Functions	Complete	Cost	Search Tree	Avg. Height	Avg. Width
2	8	8	12	304	3.48	3.50
3	68	68	268	339,328	6.44	3.64
4	3,904	3,284	23,702	4,295,000,870	12.81	3.85

(a) Representative Functions

Function Class	Max Input Completed	Cost	Search Tree	Avg. Height	Avg. Width
AND	6	15	41125	13.79	3.60
NAND	6	15	17585	13.30	3.64
OR	6	15	18855	13.24	3.64
NOR	6	15	40902	14.74	3.60
XOR	3	10	32122	7.78	3.63
XNOR	3	10	29240	7.71	36.3
MAJ	4	12	101643	7.97	3.62
MUX	5	31	436885	11.89	3.84

(b) Classes of Functions

Figure 5.22: Results with Complemented Inputs and Building Block Set {AND2, OR2, NOT}

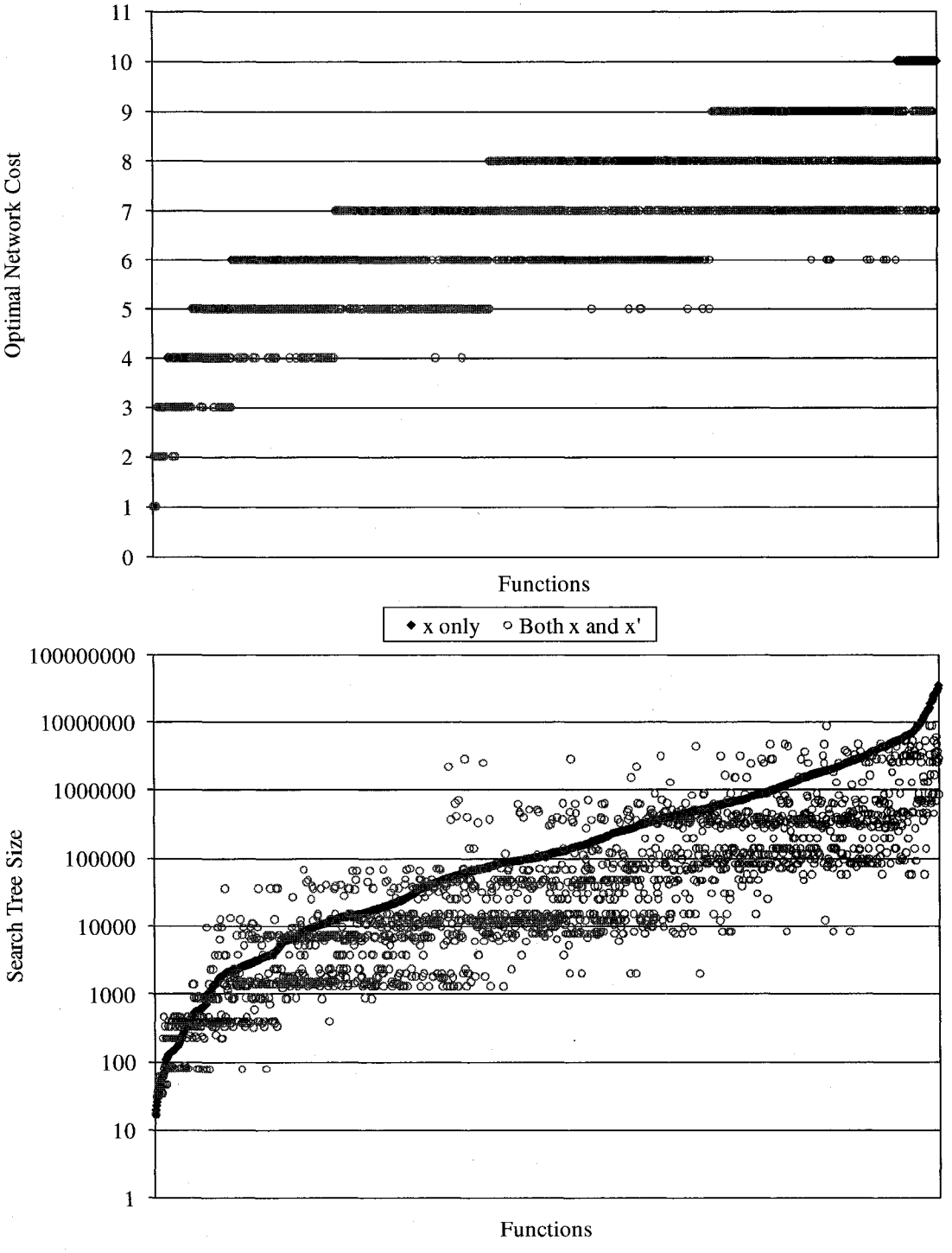


Figure 5.23: Graphs of Results with Complemented Inputs and Building Block Set {AND2, OR2, NOT}

When the results here are compared to those presented in Figure 5.17 where the building block set {AND2, OR2, NOT} was used with only uncomplemented inputs to the network, similar observations can be made to those observed when {NAND2} gates were used. The cost of an optimal network is reduced for 90% of the functions. The average decrease is approximately 1.4 gates. The reduction in the cost of the network helps to reduce the size of the search tree in this case too. The size of the search tree is reduced by as much as 100%.

Further reductions in the search are made when the NOT gate is removed from the building block set. The first graph in Figure 5.25 shows that the cost of the optimal network will increase for some functions while decreasing for other functions compared to the case when complemented input are not provided to the network. Despite these increases, the second graph in Figure 5.25 shows that for the vast majority of the functions the search can be reduced by adding the complemented inputs and removing the NOT gate from the building block set. The additional reduction in the size of the search tree can be attributed to a reduction in the options for gate types.

Inputs	Functions	Complete	Cost	Search Tree	Avg. Height	Avg. Width
2	8	8	12	115	3.09	2.69
3	68	68	272	653,202	6.07	2.75
4	3904	3,196	23,560	4,665,373,338	13.08	3.03

(a) Representative Functions

Function Class	Max Input Completed	Cost	Search Tree	Avg. Height	Avg. Width
AND	6	15	7,727	14.45	2.73
NAND	6	15	8,735	14.20	2.74
OR	6	15	8,812	14.53	2.74
NOR	6	15	6,990	15.31	2.73
XOR	3	12	305,105	9.32	2.84
XNOR	3	12	274,507	9.13	2.86
MAJ	4	12	28,608	7.45	2.77
MUX	5	31	146,842	12.63	3.00

(b) Classes of Functions

Figure 5.24: Results with Complemented Inputs and Building Block Set {AND2, OR2}

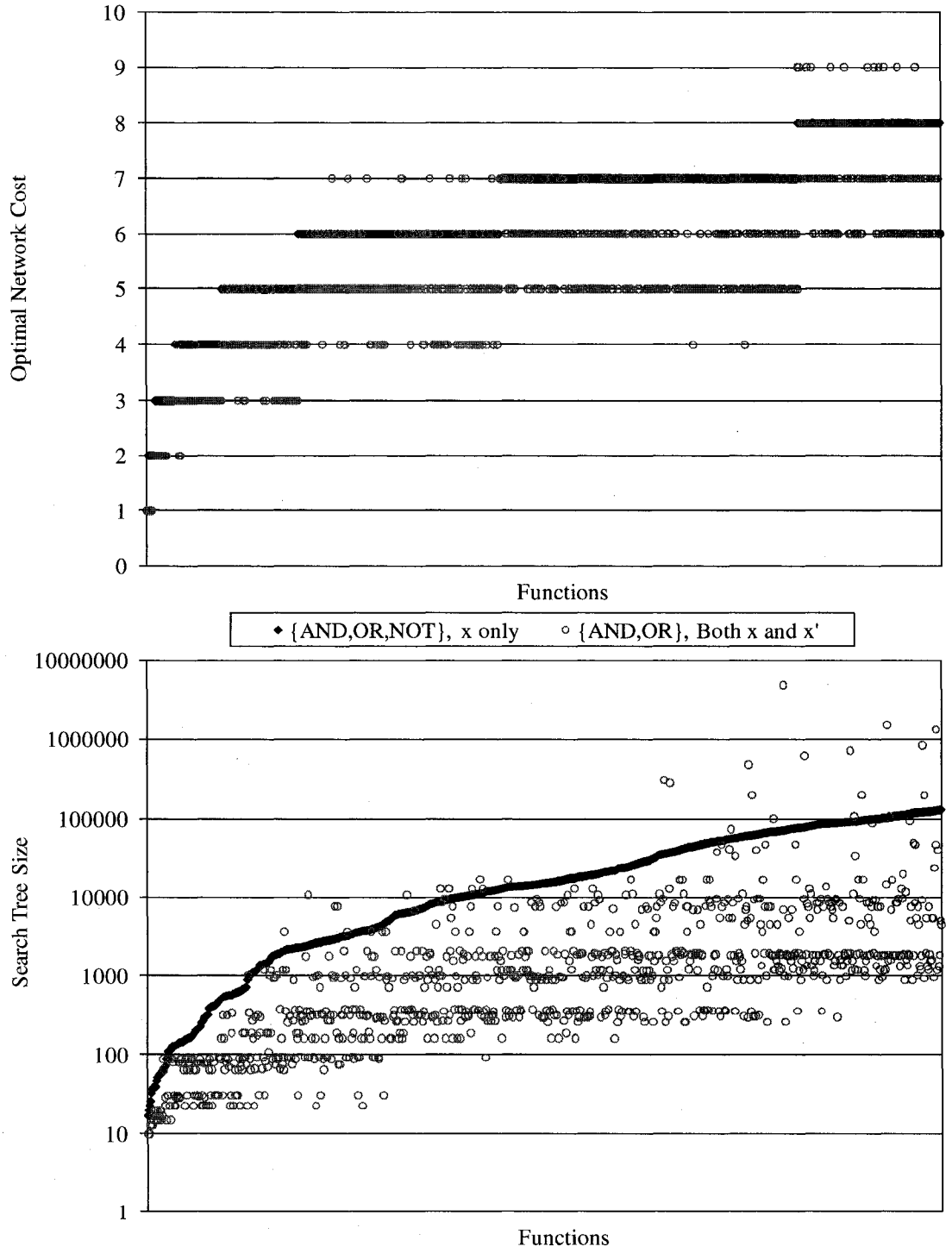


Figure 5.25: Graphs of Results with Complemented Inputs and Building Block Set {AND2, OR2}

5.7 Previous Work on Exact Synthesis

Now that we have presented our algorithm BESS in detail and given results for several variations, we can relate this work to previous work in the area of exact synthesis. These algorithms, as described in Chapter 1, can be divided into three groups according to what method they use to approach synthesis. We will discuss the previous work by providing details about the algorithms within these categories. We will then consider some of these algorithms in more detail and compare their results to what we have presented thus far. We expect to see improvements in the size of the networks completed due to improvements in the algorithm and improved computer technology.

5.7.1 Functional Decomposition

One method for solving the problem of exact synthesis is by using functional decomposition. The decomposition of a logic function $f(x_1,\ldots,x_n)$ is the process of identifying a set of functions $\{h, g_1,\ldots, g_m\}$ such that $f(x_1,\ldots,x_n)=h(g_1(X_1),\ldots,g_m(X_m))$, where $X_1\cup\cdots\cup X_m=\{x_1,\ldots,x_n\}$. A decomposition is **disjoint** if the sets $X_1,\ldots,X_m$ form a partition of the variables $\{x_1,\ldots,x_n\}$. A decomposition where the intersection of at least one pair of X_i and X_j is not empty is a **non-disjoint decomposition**.

Logic synthesis is completed using functional decomposition by performing repetitive decompositions of the form $f(x_1,\ldots,x_n)=h(g_1(X_1),\ldots,g_m(X_m))$ until all functions $\{h, g_1,\ldots, g_m\}$ are contained in the specified set of building blocks.

Ashenhurst was the first to consider the problem of designing multi-level networks of Boolean functions with his work on disjoint decomposition [Ashenhurst 59]. In this work, Ashenhurst presents the theoretical result which describes the criteria that must be satisfied for a Boolean function to have a "simple" disjoint decomposition of the form $f(x_1,\ldots,x_n)=h(X_1, g(X_2))$ where X_1 and X_2 form a partition of the set $\{x_1,\ldots,x_n\}$. He then goes on to present a method for detecting such decompositions. His algorithm uses *decomposition charts* to identify the decompositions that exist for a given Boolean function including incompletely specified functions. He concludes his work by defining complex decompositions in terms of simple decompositions. This work culminates in a procedure which determines the complex decomposition structure (the tree network) for a function based on its set of simple decompositions using the decomposition charts.

Curtis [Curtis 61] extends the theoretical and algorithmic work of Ashenhurst to the case of decompositions of the form $f(x_1,\ldots,x_n)=h(g_1(X_1),\ldots,g_m(X_1),X_2)$ where X_1 and X_2 form a partition of the set $\{x_1,\ldots,x_n\}$. This decomposition is used to produce "generalized" tree networks. The theoretical

results presented give criteria for determining if such decompositions exist. The corresponding algorithm illustrates an extension of Ashenhurst's method using decomposition charts to identify the decompositions that exist for a given function. In an attempt to solve the optimality part of the synthesis problem, Curtis outlines a procedure for determining the "best" tree network based on cost bounds associated with the functions used in each available decomposition [Curtis 59]. The generation of the best tree network however is based on a greedy heuristic of selecting the decomposition with the least estimated cost as the one most likely to produce an economical network.

In both the work of Curtis and Ashenhurst, synthesis is performed by a series of specific decompositions. Decompositions of this form are repeated on portions of the resulting structure until they are no longer possible. There is no pre-specified set of building blocks which are used for the decomposition. This makes it difficult to assign cost functions to the resulting networks.

In [Roth 60], Roth extends the functional decomposition work of Ashenhurst and Curtis to a synthesis framework where a set of building blocks and related costs are part of the problem specification. Roth presents an algorithm for finding a minimum-cost tree network which implements a given Boolean function in terms of a given set of logic gates. Each gate contained in the set of building blocks used for constructing the network has a cost value associated with it and the cost of a network is the sum of costs over all gates used in the network. Roth moves away from the decomposition charts used by Ashenhurst and Curtis to a more compact normal form representation for Boolean function. A change in format for representing Boolean functions also requires a new way for finding decompositions. A calculus for performing logic operations is developed and a projection operator is created to determine the decompositions that exist.

To this point, algorithms for exact synthesis using functional decomposition only produce networks which have a tree structure. The algorithm presented by Karp et. al. [Karp 61] makes the leap to general networks. The algorithm described in [Karp 61] produces a multi-level network using a set of logic gates with unlimited fan-in and fan-out as building blocks. However, there still remains one restriction on this algorithm. Only "vertex functions" are allowed as building blocks. A vertex function is defined to be a Boolean function which evaluates to one of its values (1 or 0) for exactly one assignment of the input variables and the opposite value for all other values. The Boolean functions AND and OR are examples of vertex functions. The authors note that vertex functions have useful properties that allow for gains in the efficiency of the program but in practice little is lost by restricting to only this set since most primitive gates used for synthesis are vertex functions. In a later work [Roth 62] this vertex function restriction is removed and the same algorithm is presented for any set of building blocks. Once again a key to this algorithm is the representation of incompletely specified Boolean functions.

In [Karp 61] and [Roth 62] on- and off-arrays are used to describe incompletely specified functions and these on- and off-arrays are represented in a normal form similar to [Roth 60]. The algorithm is a branch-and-bound algorithm which searches systematically through valid decomposition sequences to determine

the sequence which will produce a minimum cost network. Each branch is a choice of existing decompositions and the algorithm bounds when (a) a network is completed, (b) the cost of the network exceeds an upper bound, or (c) a cyclic network is generated. Using an implementation of the proposed algorithm the authors were able to obtain results on several functions ranging from three to five inputs. However, they soon discovered that the size of the search space quickly becomes a problem. "A search of over five hours failed to improve the implementation. It was determined, however, that the search was far from complete. Further, it was determined that almost all the search time was spent working on an array of three variables or less."

Schneider and Dietmeyer [Schneider 68] further extend the previous work to handle complex logic operations as building blocks. They use a goal-based method similar to those from game playing algorithms to make the decomposition choices but stop searching once the first network is found. The authors admit that the algorithm described is a heuristic method for the optimal problem since only a small portion of the entire search space is explored. However, they go on to mention that using this method to exhaustively search the solution space is possible and will result in the discovery of the minimum cost circuit. Doing this of course will "cause exponential increases in the amount of time and effort expended by the algorithm." [Schneider 68]

Lawler [Lawler 64] approaches the problem of optimal multi-level synthesis by extending results from two-level synthesis to multi-level synthesis. Lawler's approach is to generalize the idea of prime implicants from two-levels to n-levels. Using the new definition of prime implicants, he extends the prime implicant based two-level synthesis to the multi-level case. Since his algorithm minimizes the number of literals in the multi-level expression, the results of this algorithm are tree networks where the number of connections to the inputs is minimized. His approach is a unique direction that is different than any attempts at the problem to this point.

From this set of previous work we can see that the ideas of functional decomposition originally introduced by Ashenhurst and Curtis can be used as a tool to perform exact synthesis. We have seen that this method works for any type of logic function used as building blocks including simple functions as in [Karp 61] or more complex functions as in [Schneider 68]. We have also seen that completely-specified as well as incompletely-specified functions can be optimally synthesized using these techniques. As expected the intractability of the optimization problem became evident when implementations of the algorithms were executed on larger examples. The advantage of the branch-and-bound nature of these algorithms is that a network implementing the given function (and often the optimal one) is found quickly. The remainder of the time is spent searching through the solution space proving that the network is optimal.

5.7.2 Network Enumeration

Network enumeration is a structural method for approaching the problem of exact synthesis. It is based on the fact that all possible networks with g gates can be enumerated. Using such an enumeration the space

of g gate networks can be searched for one that generates the desired function. If this search process is repeated, beginning with $g = 1$, for an increasing number of gates then the first network found during the search is an optimal network with respect to a cost function based on the number of gates in the network. If optimization with a second criterion is desired then all networks with g gates must be searched before the algorithm can complete to guarantee optimality.

5.7.2.1 Explicit Enumeration Algorithms

There are three main results which use explicit enumeration to perform exact synthesis. Hellerman [Hellerman 63] was the first to use this method. His goal was to provide optimal {NAND} and optimal {NOR} networks for all Boolean functions on three or fewer variables. Therefore network enumeration provides an efficient way to generate all of these networks at once. He is able to provide optimal {NAND} networks and optimal {NOR} networks for all but one representative function from the P-equivalence class for functions with 3 or fewer inputs. He uses a cost functions based first on the number of gates in the network and second on the number of connections in the network. He also restricts the number of fan-in and fan-out of the gates to 3. With this enumeration algorithm he is able to complete all functions which require networks with 7 or fewer gates. The function $x_1 \oplus x_2 \oplus x_3$ requires 8 gates so an optimal network for this function was not proven optimal.

Smith [Smith 65] extends the work of Hellerman by presenting a similar table of results for networks where both complemented and uncomplemented inputs exist. The same cost function is used; however the fan-in and fan-out restrictions are removed. With this enumeration algorithm, networks for all functions from the P-equivalence class with 3 or fewer variables were completed. He found that only 18 networks structures exist for all 80 functions. Rearrangement of the input values in these structures provides implementations for each of the functions. The largest of these 18 networks requires 5 gates.

Finally, Drechsler and Günther present an updated version of this structural enumeration algorithm in [Drechlser 98]. They are able to extend the results of Hellerman past all 3-input functions to include some 4-input functions and some larger. They did this through the addition of extensive pruning techniques to minimize the search space. They still were not able to complete some 4-input functions, however. Their work focused on {NAND} networks with a fan-in restriction of 2 but no fan-out restriction. In addition to {NAND} networks, they also present results using AND, OR, and NOT gates. The main results from [Drechsler 98] can be compared to those given in Figure 5.1. They found that this method was only applicable to networks with up to 12 gates.

Some of the improvement seen in our results over these earlier results can be attributed to the improvement in computer technology over time. However, a comparison (Figure 5.26) of an implementation of the algorithm from [Drechsler 98] with BESS shows that there are more factors involved. Our experiments showed that the limiting factor on finding an optimal network for both algorithms was the number of gates needed for the optimal network. For the network enumeration

algorithm, any function which required more than 9 gates was not able to be completed within the specified one hour time limit. For BESS, this limit was raised to 13 gates. The limit of 9 gates prohibits the enumeration algorithm from completing those four-input functions that require a larger number of gates.

Inputs	Network Enumeration			BESS		
	Completed Functions	Avg. # Connections	Avg. Time	Completed Functions	Avg. # Branches	Avg. Time
2	8	32	0 s	8	3	0 s
3	67	337,610	46 s	68	37	0 s
4	73	22,500,000	3400 s	207	44,415	64 s

Figure 5.26: Comparison of a Network Enumeration with BESS

These results show that BESS is able to find and prove the optimal network in less time than the other method. Further comparison of the two algorithms reveals two advantages our algorithm has over this enumeration method. First, the search space that it searches is on average smaller. BESS only has to search the set of all networks that implement the desired function whereas the enumeration algorithm must search the entire set of all networks. Second, a smaller portion of the search space is repeated BESS. While repetition does still occur in the search, the enumeration algorithm must repeat the entire search completed for the previous iteration.

5.7.2.2 Implicit Enumeration Algorithms

The enumeration that is performed by the previous set of algorithms can be performed implicitly using constraint satisfaction. The set of all possible networks with n gates is represented as a universal network where all connections between gates in the network are included. This universal network is then encoded as an instance of a constraint satisfaction problem (CSP) such that a solution to the CSP will provide the edges that should remain in the network to produce the desired functions.

[Muroga 72] presents an algorithm which uses integer programming to solve the constraint problem created by the universal network. With this algorithm optimal {NOR} and {NOR, AND} networks were completed for all representative functions from the P-equivalence classes with 3 or fewer inputs. The cost function used minimizes the number gates in the network as the primary criteria and the number of connections within the network as the secondary criterion. No fan-in or fan-out restrictions were imposed on the networks. The authors discovered that networks containing 9 or fewer gates could be completed. Additional similar results were presented in [Baugh 69] [Culliney 71] [Baugh 72] [Muroga 76].

It appears that little was gained by performing the enumeration implicitly. The results presented using this method are not all that different than the results we saw with the explicit enumeration algorithms. The algorithm must search the same space so improvements can only come from the speed at which the CSP solver is able to complete the search. In addition, the same repetition will occur in this search as in the

explicit enumeration since the entire search of the previous number of gates must be repeated when the number of gates is increased.

A comparison (Figure 5.27) of an implementation of an implicit enumeration algorithm using a SAT solver to solve the CSP instances (presented as quantified Boolean formulas) with BESS confirms these results. Using this method, optimal results were obtained for all two- and three-input functions as well as some four-input functions when the search time per function was limited to one hour. Once again the number of gates in the optimal network provides the limiting factor for completing the search. The number of gates in a universal network determines the number and size of constraints as well as the number of variables needed in the quantified Boolean formula. Therefore the larger the universal network that must be used to find a solution the more work is required for the SAT solver. From this experiment we discovered that functions requiring more than 9 gates could not be completed in the allotted time.

Inputs	Network Encoding			BESS		
	Completed Functions	Avg. # SAT calls	Avg. Time	Completed Functions	Avg. # Branches	Avg. Time
2	8	29	0 s	8	3	0 s
3	64	4,116	99 s	68	37	0 s
4	42	69,487	3021 s	207	44,415	64 s

Figure 5.27: Comparison of Network Encoding with BESS

Whether implicitly or explicitly performed, network enumeration provides a simpler way to perform exact synthesis compared to functional decomposition. The work by Hellerman and Smith use it to generate tables of results for a given number of inputs with minimal overlap of work. [Drechsler 98] and [Ibaraki 72] show that it can be used to solve individual synthesis problems as well. The simplicity of these algorithms allow for easy change of the logic gates used as building blocks since none of the complicated decomposition criteria from the functional methods are required to be reworked. However the simplicity leads to disadvantages as well. Unlike the branch-and-bound methods, the majority of the work done here is done before a network is found. In addition, the incremental nature of the method forces a large portion of the search to be repeated each time the number of gates is increased.

5.7.3 Functional and Structural Methods

The last set of algorithms that we will examine is the set of branch-and-bound algorithms which provide the basis for the algorithm presented here. Sections 5.7.2.1 and 5.7.2.2 showed why this was our preferred method for completing exact synthesis. This section allows us to observe the results of our updates to this branch-and-bound method.

The first presentation of this algorithm was given by Davidson in [Davidson 68b][Davidson 69]. This work uses the branch-and-bound algorithm to provide optimal {NAND} networks for all representative functions from the P-equivalence class with 3 or fewer inputs. Multiple cost functions are used with

different rankings on the number of gates and the number of interconnection in the network. Results both with and without fan-in and fan-out restrictions are presented. In addition to the 3-input single output functions, Davidson provides results of the algorithm on 11 other functions as well. These functions and details of how BESS performed on these function is given in Figure 5.28. For these functions, no fan-in or fan-out constraints were used and the cost function is simply the number of gates in the network.

Function	Optimal Network	Search Space	Time
Prather: $f_1 = x_2x_3' \vee x_1x_2'x_3$ $f_2 = x_1' \vee x_2x_3'$ $f_3 = x_1x_2' \vee x_2x_3$ $f_4 = x_2'x_3' \vee x_1'x_3'$ $f_5 = x_3'$	11 gates	686	0 s
Gimpel 1: $f = x_1'x_4 \vee x_2x_3'x_4 \vee x_2'x_3'x_4'$	6 gates	102	0 s
Gimpel 2: $f = x_1x_2'x_3' \vee x_1'x_2x_3' \vee x_1'x_2'x_3 \vee x_1x_2'x_4$	7 gates	324	1 s
2 out of 5: $f = x_1'x_2'x_3'x_4x_5 \vee x_1'x_2'x_3x_4'x_5 \vee x_1'x_2'x_3x_4x_5'$ $\vee x_1'x_2x_3'x_4'x_5 \vee x_1'x_2x_3'x_4x_5' \vee x_1'x_2x_3x_4'x_5'$ $\vee x_1x_2'x_3'x_4'x_5 \vee x_1x_2'x_3'x_4x_5' \vee x_1x_2'x_3x_4'x_5'$ $\vee x_1x_2x_3'x_4'x_5'$	27 gates*	17,851,755	Stopped after 24 hrs
MM: $f = \left(x_1x_2 \vee x_1x_4x_7x_8 \vee x_1x_4'x_5x_6 \vee x_1x_5x_7x_8 \vee x_3x_4x_7x_8 \vee x_2'x_3x_4'x_5x_6 \vee x_2'x_3x_5x_7x_8\right)'$	11 gates*	96,607,727	Stopped after 24 hrs
sandd: $f_1 = x_3 \vee x_4' \vee x_5$ $f_2 = x_3 \vee x_4' \vee x_5'$ $f_3 = x_1'x_3' \vee x_1'x_4' \vee x_2'x_3' \vee x_2'x_4' \vee x_5$ $f_4 = x_3 \vee x_4'$	8 gates	270	0 s
4-input Decoder	32 gates	195	0 s
Stifel 1: $f_1 = x_1x_5$ $f_2 = x_2x_5$ $f_3 = x_4x_8$ $f_4 = x_4x_9$ $f_5 = x_3x_5 \vee x_3x_7 \vee x_3x_8 \vee x_3x_9$ $f_6 = x_1x_5 \vee x_4x_8 \vee x_4x_9 \vee x_3x_6$ $\vee x_3x_7 \vee x_3x_8 \vee x_3x_9$	16 gates*	79,318,582	Stopped after 24 hrs
1-stage adder: $f_1 = x_1 \oplus x_2 \oplus x_3$ $f_2 = x_1x_2 \vee x_1x_3 \vee x_2x_3$	8 gates	437	0 s
2-stage adder:	43 gates*	645,870	Stopped after 24 hrs
3-stage adder	57 gates*	29,655	Stopped after 24 hrs

Figure 5.28: Results of BESS on Functions from [Davidson 68b]

Some improvements to Davidson's branch-and-bound algorithm were suggested by Nakagawa, Lai, and Muroga [Nakagawa 89]. Using this improved algorithm, the authors were able to obtain {AND, OR} networks for all function from the NPN-equivalence class with 4 or fewer variables [Culliney 79]. They

used a cost function based first on the number of gates in the network and second on the number of interconnections in the network. They did not restrict the fan-in or the fan-out of the gates and allowed for complemented inputs to the network. The maximum number of gates required for a network was 9; this network implemented the function $x_1 \oplus x_2 \oplus x_3 \oplus x_4$. The network found for this function contains the minimum number of gates but it was not proven that it contained the minimum number of interconnections. Thus this algorithm can only find optimal networks containing 8 or fewer gates. When running BESS using the same constraint we are able to reproduce the results obtained in this work, plus an optimal network for the function requiring nine gates was also obtained. This algorithm was also used to provide the results in [Lai 74] for the same functions shown in Figure 5.28.

5.7.4 Previous Work Discussion

A comparison of the branch-and-bound method with both the explicit and implicit enumeration methods shows that this method performs the best. The branch-and-bound method searches a smaller solution space and less of the search is repeated. It also has the flexibility of changing building blocks and cost functions without expanding the search space.

A comparison of our branch-and-bound algorithm to a variety of previous algorithms shows that an update of the algorithm has allowed us to extend the results of previous work. We were able to expand the maximum network size from twelve to sixteen gates under the same constraints. This improvement was achieved through a combination of additional pruning of the search space and an increase in the speed with which the algorithm is able to search through this space.

5.8 Summary

In this chapter we discussed variations that can be made to the original algorithm presented in Chapter 3. We described the changes required within the algorithm for each variation and then provided experimental results with this variation. Insights into how the properties of the optimal network can effect the search performed by the algorithm were gained through an analysis of the results. We learned that tighter restrictions on the fan-in and fan-out sets of the gate nodes caused the cost of the optimal networks to increase, which resulted in longer search times for the algorithm as well. Larger cost networks also resulted when a level restriction was placed on the network but the smaller solution space in this case resulted in shorter search times for the algorithm. The number of gates in the optimal networks remains the same when additional parameters are added to the cost function, however the search time increases since more of the solution space must be searched. Adding complemented inputs to the input set of the network results in smaller cost optimal networks and this implies shorter search times for the algorithm. Finally, we saw that increasing the number of building blocks available will reduce the cost of the optimal networks but the increase in the gate choices results in longer search times for the algorithm.

We concluded this chapter with a discussion of previous work. An overview of three categories of exact synthesis algorithms was presented. We then used the results presented here to compare previous algorithms for exact synthesis to BESS.

Chapter 6

Near-optimal Results

6.1 Motivation

The experimental results from Chapters 4 and 5 showed that the exact synthesis algorithm, BESS, can find optimal networks for functions with up to 16 gates in a reasonable amount of time. Given enough time, the algorithm can find and prove optimal networks for any function. However, as the search space discussion in Section 4.4 showed, this will not be feasible as the size of the networks increases. Figure 4.40 showed that very few of the synthesis benchmark functions that were attempted were completed due to the large search space. In this section we will relax the optimality constraint of the algorithm to allow the algorithm to complete these larger networks.

The nature of the algorithm lends itself very well to generating near-optimal networks. An initial complete network is found very quickly by the search. Additional networks are completed only if they have a cost lower than the cost of this initial network. Therefore the search can be stopped at any time and the smallest complete network found thus far can be used as the implementation of the function.

The table in Figure 6.1 shows that the first network found by the algorithm occurs very early in the search and that in many cases this initial network is optimal or very close to optimal. Since a small initial network leads to a smaller search space, the heuristic and pruning techniques discussed in Chapter 3 help to produce an initial network that is close to optimal. The data show that on average, the initial network cost is within 2.9 gates or 27% of optimal and the optimal network is found within the first 38% of the search. Thus, the networks found early in the search can be used as a close-to-optimal result for synthesis.

Function Group	Total Optimal Cost	Average Optimal Cost	Total Number of Branches in Search Tree	Total Initial Cost	Average Initial Cost	Number of Branches at Initial Network
2-input Reps from P-Equiv Class	24	3.00	31	24	3.00	31
3-input Reps from P-Equiv Class	405	5.96	2,080	419	6.16	572
4-input Reps from P-Equiv Class	37,936	9.72	65,428,000	49,298	12.63	78,092
2-7 input AND	42	7.00	601,235	42	7.00	33
2-6 input OR	45	9.00	1,006,037	45	9.00	45
2-4 input XOR	24	8.00	6,055	26	8.67	36
2-5 input MAJORITY	31	7.75	66,692,278	44	11.00	69
3-6 input MULTIPLEXER	50	8.33	20,432	63	10.50	100

Figure 6.1: Algorithm Results Comparing the Initial Network with the Optimal Network

While the results from Figure 6.1 show that the average initial cost is within 27% optimal, the range is actually between 0% and 170% of optimal. As with any near-optimal result, the ability to control this range is desired. This control would allow us to guarantee that the network found by the algorithm is within a set number of gates or a fixed percentage of optimal. This control can be accomplished by changing the bounding technique and allowing the algorithm to fully complete the search rather than stopping the algorithm partway through the search. The parameters that can be used to perform this suboptimal control will be given in Section 6.2 along with experimental results of these methods.

Near-optimal results for larger functions including those benchmark functions which were previously too large to be completed optimally are presented in Section 6.2.5. We then use these results to test a well-established multi-level synthesis heuristic, ABC, in Section 6.3. We use both the optimal and near-optimal results to test the networks produced by ABC. This comparison allows us to evaluate how well ABC is able to produce near optimal networks.

6.2 Near-optimal Methods

The goal for the near-optimal methods described here is to reduce the amount of the space that must be searched in order for the algorithm to complete synthesis on as many functions as possible. There are three ways that near-optimal results can be produced by the algorithm with this goal in mind. The first method will simply stop the search after a set amount of time and then use the lowest cost network found by the algorithm at that time. This method provides a guarantee for the amount of time required by the search, but the distance that the network will be in relation to optimal cannot be determined.

The second method lowers the upper bound of the cost function by a set amount every time a new network is found. This will cut the amount of the solution space that the algorithm must search since partial networks can be pruned earlier due to their costs. When the algorithm finishes the search, the

network will be within a known number of gates of optimal because all networks with cost less than the cost bound will have been searched.

The third method also lowers the bound for the cost function by a fixed amount. However, in this method, the lowering of the cost bound will be based on the amount of the search that has been completed. Therefore, as the search tree grows larger, the cost bound will be made smaller and so more of the search can be pruned. This method will provide a trade-off between a reduction in the solution space searched and an increase in the distance the final network will be from optimal.

6.2.1 Test Functions

The following is a description of the functions that will be used to evaluate the variations of the algorithm to produce near-optimal results. The variations will be tested using both large and small functions. This will ensure that the near-optimal methods will be tested on functions where the search has been completed and optimal results are known as well as on functions where the search space was previously too large to be completely searched and an optimal network remains unknown.

The first two sets of functions will be drawn from those functions for which BESS is able to complete the search and produce optimal networks. The **Equiv** set is composed of all 3,904 P-representative functions with two through four inputs. The **Class 1** set includes the AND functions with 2 – 7 inputs, the OR functions with 2 – 6 inputs, the XOR functions with 2 – 4 inputs, the MAJORITY functions with 2 – 5 inputs, and the MULTIPLEXER function with 3, 5, and 6 inputs. Since an optimal network has been found for each of these functions, we can use the optimal results to determine the exact distance a near-optimal result is from optimal. We can also use the details of the search for these optimal networks to compare the savings that are gained by relaxing the optimality constraint.

The second two sets of functions will be drawn from those functions for which an optimal network was not found by BESS. The **Bench** set is composed of the 18 benchmark functions from Figure 4.40. The **Class 2** set includes 13 functions: functions from the AND class with 8 – 9 inputs, functions from the OR class with 7 – 9 inputs, functions from the XOR class with 5 - 7 inputs, functions from the MAJORITY class with 6 - 8 inputs, and functions from the MULTIPLEXER class with 8 and 9 inputs. BESS did not complete the search for these functions so a comparison of the near-optimal and optimal networks and searches cannot be made. However these function sets will allow us to evaluate the increase in the size of the functions that can be synthesized using the near-optimal versions of the algorithm.

6.2.2 Near-optimal Method: Time Constraint

In the first near-optimal method, the search will simply be stopped after a fixed amount of time. The last network found by the algorithm will be the network with smallest cost chosen for implementing the function. The amount of time the algorithm is allowed to search is a tunable parameter for this method. Assigning different values to this parameter will determine which value gives the best results with the least

amount of work. First, the algorithm will be stopped as soon as the first network is found. Then the algorithm will be stopped after 1, 5, 10, and 24 hours.

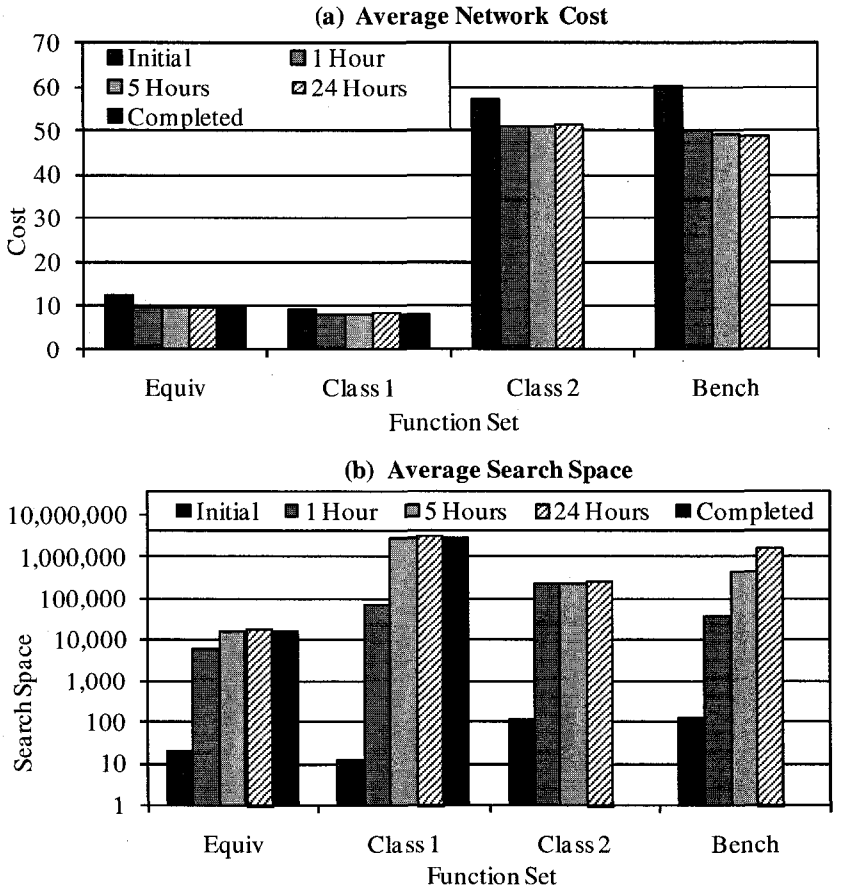


Figure 6.2: Time Constraint Method

Figure 6.2 gives the results of the experiments using these parameters. Graph (a) in Figure 6.2 shows the decrease that occurs in the average cost as the time bound increases while (b) shows the increase in the size of the search as the time bound increases. Together these two graphs show the trade-off between the cost of the network found by the algorithm and the size of the search that must be completed to produced this network.

For the first two function sets, the algorithm completed an optimal network for every function within the first hour. However, the algorithm required between 1 and 5 hours to complete the search in some cases. The time spent by the algorithm after the optimal network is found is only used to prove optimality of the network.

A similar pattern was found for the Class 2 set of functions. No improvements were made to the cost of the networks past the first hour of the search. On the set of benchmark functions, improvements to the cost of the networks were made for at least one function within each time interval. In this case, the longer the

algorithm is allowed to run, the smaller the cost of the network becomes. However, the improvements tend to plateau. For 63% of the functions, at least one improvement in the network cost for the function was made within the first hour of the search. Between the first and 24th hour however, only 11% of the functions saw an improvement. There are two possible reasons for this plateau. The first is that the network found is optimal and the remainder of the search will be used to explore the remaining solution space proving that no network has a smaller cost. The second is that a smaller network does exist for the function but this network has a different structure than the current known network and therefore will not appear until much later in the search.

6.2.3 Near-optimal Method: Lowering the Cost Bound

A second method for producing near-optimal results is based on modifications to the cost bound. When a complete network is found by the SynthesizeNetwork procedure of the algorithm (Figure 3.21), the global variable UpperBound is set to the cost of the complete network. Then for the remainder of the search, all networks must have cost less than this value UpperBound. To reduce the amount of the solution space that is searched, the algorithm can artificially lower this UpperBound value below the cost of the network each time a complete network is found. This will cause some partial networks to be pruned earlier in the search due to the lower value on the cost bound. Thus the overall search will be reduced. In addition, when the algorithm completes we will be able to determine the distance that the network is from optimal. For example, if the value UpperBound is set to ($Cost_{Network}$ - 5) then the cost of the final network found by the algorithm will be within 5 gates of optimal. This is the case since all networks with lower cost were searched by the algorithm. Using this lower bound on the cost of the network and the cost of the last network found by the algorithm as an upper bound, a cost interval can be determined for each function.

The parameter used to control the size of the search and the optimality of the result in this method is the value by which the UpperBound is reduced from the cost of the found network. This type of bounding modification can be done either with a fixed number of gates or with a percentage of the cost of the network. In our experiments we tried several versions of each type.

First, we lower the cost bound by a constant of c = 5, 20, 50, and 100 gates. The summary of these results are given in Figure 6.3. The algorithm was allowed to run for 3 hours on each function instance. For each function, the network found by the search is within the value c of optimal. As the value c increases, more of the search space can be pruned, creating smaller search trees.

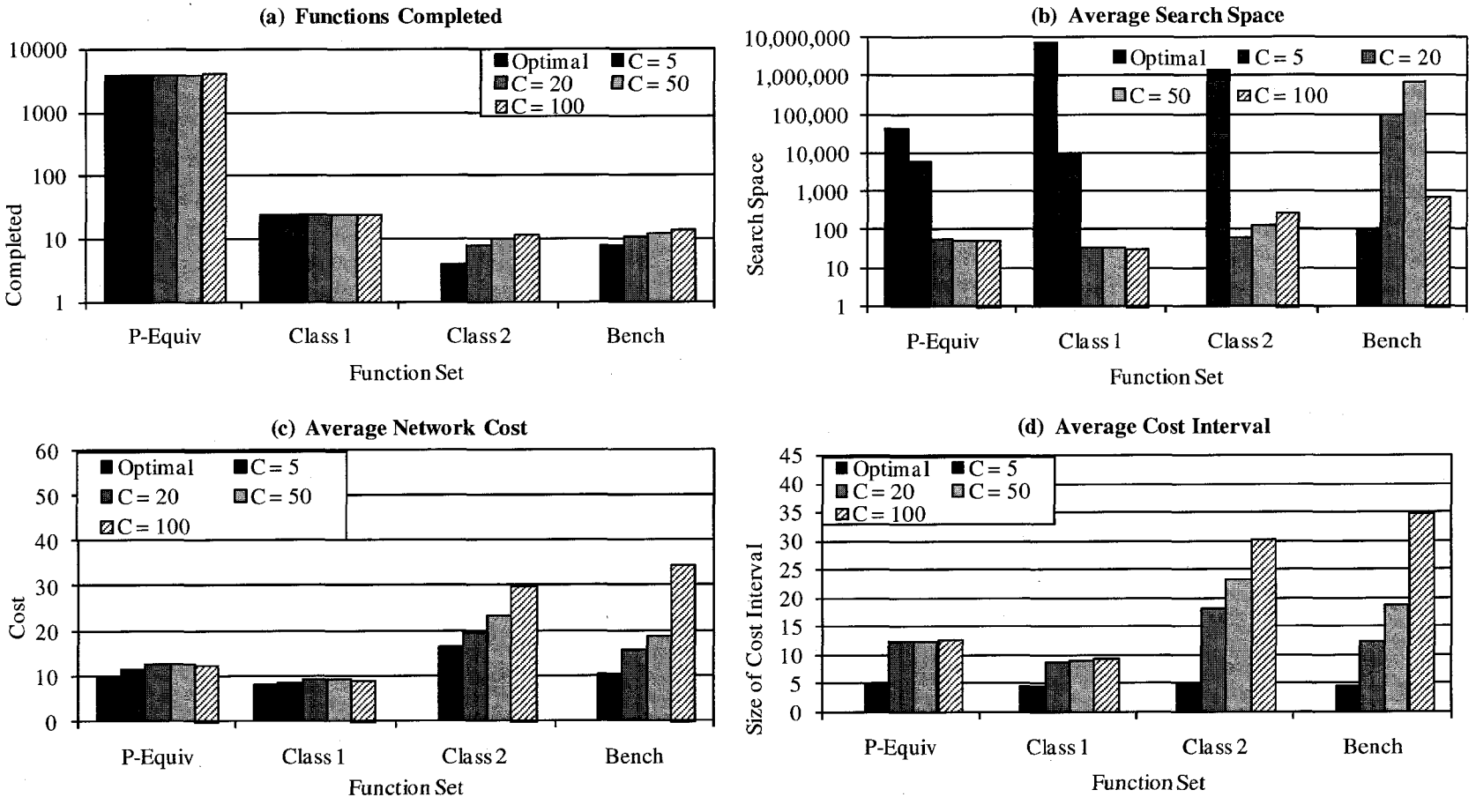


Figure 6.3: Constant Lowering of UpperBound

Graph (a) of Figure 6.3 shows the increase in the number of functions on which the search is completed as the value of c is increased. The first two sets contain functions which were completed by the optimal version of the algorithm. Therefore no change is seen here. In the second two sets, the increase in the number of functions completed is seen. We found that the minimum value of c required in order for the synthesis of one new function to be completed was 2 and that at the maximum reduction of 100 gates, 24 of the 31 functions with unknown optimal cost could now be completed.

A decrease in the size of the search space explored by the algorithm results as the constant value is increased. This is shown in graph (b) of Figure 6.3. The search space can be reduced to only 7% of it original size by lowering the cost bound 5 gates below the actual cost of the network. When this is increased to 100 gates, the search space is lowered to 0.06% of the search space size when the optimal search is performed. The decrease in the amount of solution space searched produces the increase in the number of networks completed by the search as seen in graph (a). The increases seen in (b) for the sets of functions with unknown optimal cost are due to the increases in the number of completed functions.

Graphs (c) and (d) in Figure 6.3 show the penalty incurred for the saving gained in the search as the value of c increases. An increase in the constant c results in an increase in the cost of the networks found by the near-optimal versions of the algorithm. For the first two sets of functions, the cost of the networks increases to 17% above optimal when $c = 5$. At $c = 100$, the costs increase to an average of 30% above optimal. In some cases, the optimal cost interval shown in graph (d) is lower than the value of c. This is due to the fact that the lower bound of the optimal cost interval must be greater than 0. Therefore, if c is greater than the current cost bound, the bound will only be lowered to a value of 0. The result is then an optimal cost interval smaller than c.

The trade-off between the size of the cost interval and the size of the search tree is quite evident by comparing graphs (b) and (c). To maximize the number of networks that can be completed by the algorithm, the constant value must be increased to a value of 100. While the search can be performed very quickly with c at this value, the costs of the networks completed tend to be further from optimal.

A second bounding modification for this method can be performed by reducing the cost bound by a percentage of the network cost. Here the value UpperBound will be set to $(\text{Cost}_{\text{Network}} \times (1 - \text{p}))$ where p is some percentage. Thus, the size of the reduction of the network is based on the current size of the network. This modification will reduce the larger networks enough so that the search can be completed but will not increase the cost of the smaller networks unnecessarily. Again this modification was tested with different values for the parameter p. Results from these experiments are given in Figure 6.4. Just as in the previous results, an increase in the percentage p allowed for more of the search space to be pruned, creating smaller search trees while increasing the optimal cost interval.

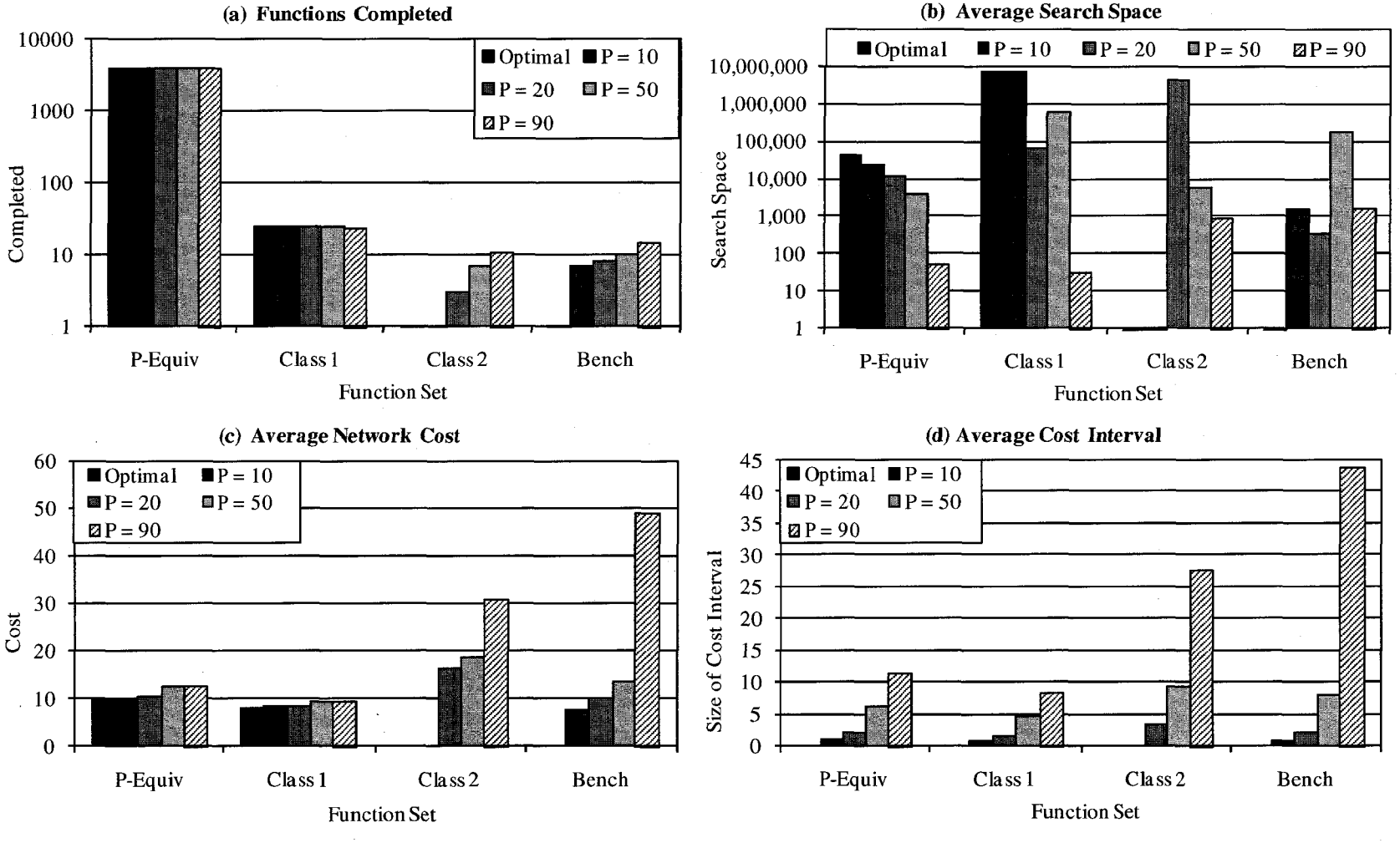


Figure 6.4: Percent Lowering of UpperBound

Graph (a) in Figure 6.4 shows the increase in the number of functions on which the search is completed. The value $p = 20\%$ was the minimum percentage required in order for the search to be completed on at least one additional function. At the maximum value $p = 90\%$, 25 out of the 31 functions were completed from the sets Class 2 and Bench.

The graph in Figure 6.4 (b) displays the correlation between an increase in the percentage p and a decrease in the size of the search performed by the algorithm. At the smallest percentage, $p = 10\%$, the search space of the first two sets of functions reduces to 28% of optimal. At the maximum value, $p = 90\%$, the search space for these sets is reduced to only 0.06% of optimal. The large reduction in the search space allows for the increase in the number of functions completed from the function sets Class 2 and Bench.

The increase in the search space seen in the Class 1 and Bench function sets is due to an increase in the cost upper bound for some functions as the percentage increases. A higher cost bound requires more of the solution space to be searched. For example, the initial network found by the algorithm for the 5-input majority function requires 27 gates. When $p = 20\%$ a second network is found containing 16 gates and the final value for UpperBound is 12.8 based on this network. However, when $p = 50\%$, UpperBound is set to 13.5 following the completion of the initial network containing 27 gates. In this case, the second network containing 16 gates will not be found, but the value UpperBound is larger than when $p = 50\%$ so more of the solution space must be searched.

Graphs (c) and (d) in Figure 6.4 indicate that both the cost and the cost intervals increase as the value of p increases. With the value of p at 10%, the cost of a network from the Equiv and Class 1 sets increases to only 4% above optimal. When p is increased to 90%, the average cost of a network for these same functions is increased to nearly 30% over optimal. The cost is increased more for the larger functions from the Class 2 and Bench sets.

While the trade-off between the distance from optimal guarantee and the size of the search tree remains evident from these results, they are not as pronounced as when a constant reduction was used. The average cost and cost intervals are smaller for each of the functions when a percentage value is used to modify the cost bound compared to when a constant value is used. At the same time, the search space remains small enough in both cases that approximately the same number of functions can be completed. Thus the percentage reduction keeps the suboptimality of the smaller circuits low while reducing the search space of the larger networks to allow the search to be completed.

6.2.4 Near-optimal Method: Parameterized Lowering of Cost Bound

One final method for producing near-optimal results also lowers the cost bound to help prune the search space. In this version, the cost bound is lowered based on the number of branches that have been explored in the search tree. By using the number of branches explored as a parameter in the lowering of the cost bound, finer control is possible over the trade-off between the amount of the solution space explored by the algorithm and the size of the optimal cost interval. The goal of this parameterization is to reduce the search

for each individual function in such a way that the search is small enough to be completed while keeping the optimal cost interval as small as possible.

For this near-optimal method two parameters must be tuned. The first is the value by which the cost bound is lowered. Once again, this can be either a constant value or a percentage of the current cost bound. We will use the constant values $c = 1$ and 2 and percent values $p = 1\%$ and 2%. The second value that can be altered is the number of branches that are explored before a reduction in the cost bound is performed. The smaller the value, the more often the reduction will be performed and the larger the portion of the search space that can be pruned. We will experiment with the values $b = 100$, 500, and 1000. With this formulation the value UpperBound will be reduced by a constant c or a percent p for every b search tree branches created by the algorithm.

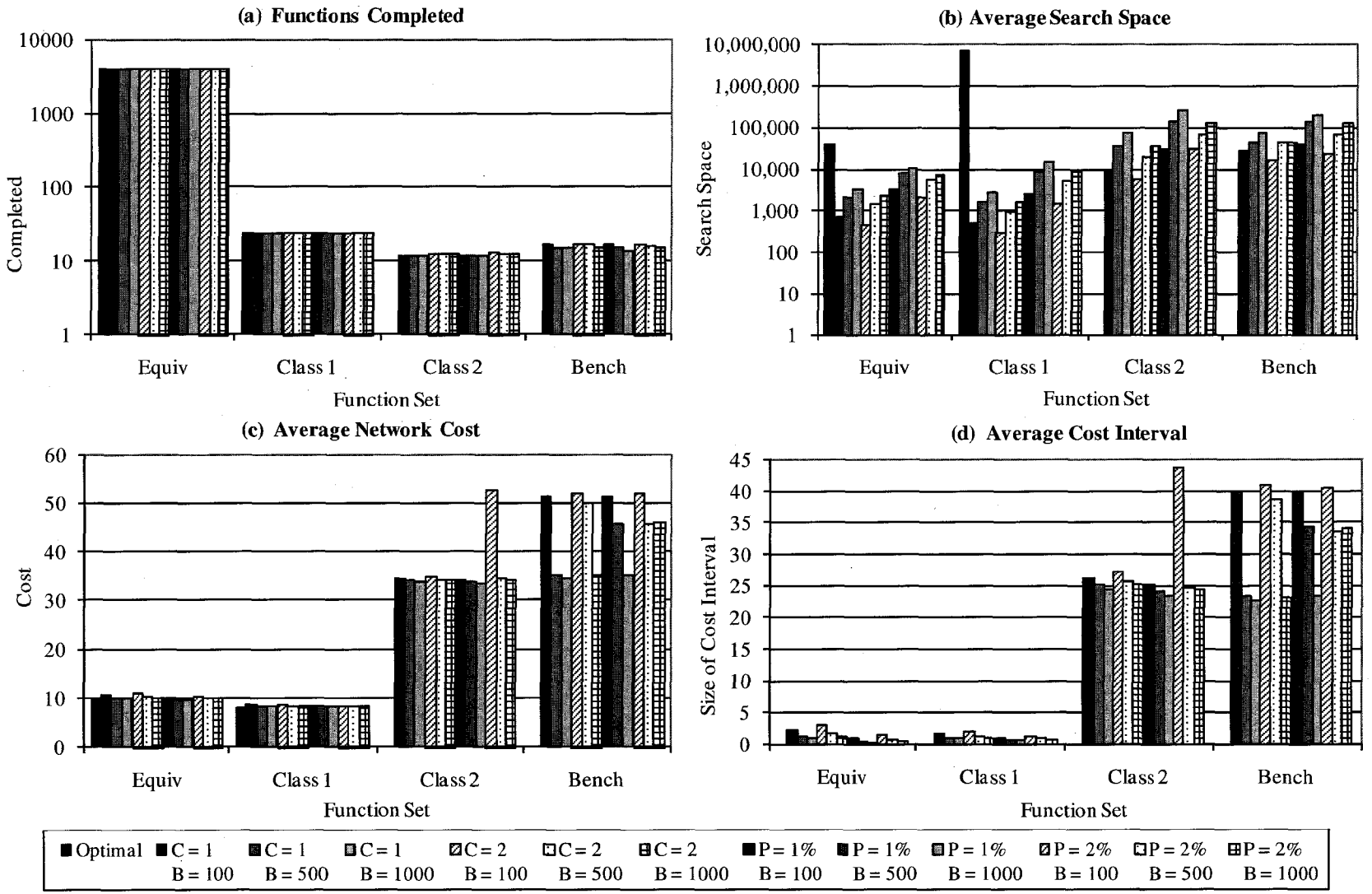


Figure 6.5: Parameterized Lowering of `UpperBound`

A summary of the results of our experiments with this near-optimal method are given in Figure 6.5. Graph (a) shows that as the values of c and p increase and the value of b decreases, the search can be completed on a larger number of functions. The maximum number of functions completed, 28 out of 31, occurs first when $c = 2$ and $b = 500$. The increase in the number of functions for which the search was completed is a direct result of the decrease in the size of the search tree. The results given in graph (b) show that when $c = 2$ and $b = 500$, the search tree size is reduced to 1.7% of the size of the search required to complete an optimal network. Larger reductions are possible by increasing c or p or lowering b. However, little effect is gained in the number of networks that can be completed.

Since the cost bound reduction is based on the number of branches completed, the cost of the network and the cost interval for the optimal network remains small for those functions where an optimal network is found and the entire search completed. Graphs (c) and (d) show that when $c = 2$ and $b = 500$, the network found by the near-optimal version of the algorithm is only an average 6% larger than the optimal network. For these same networks, the cost interval only increases to 2 gates. For those functions which were not completed optimally, the size of the average cost interval is increased to 32 gates.

This method further improves the previous near-optimal method. The suboptimality of the smaller networks are kept very low, while the suboptimality of the larger networks is increased to allow for a larger reduction in the search space. This allows for the search to be completed on a larger number of the functions.

6.2.5 Method Selection & Additional Results

6.2.5.1 Method Comparison

If all of the methods are compared together, we can evaluate the best method for completing near-optimal synthesis. The table in Figure 6.6 compares the best version from each method which completes the most functions. For each function class, the number of networks completed by the search, the average size of the search for these functions, the average network cost, and the average size of the optimal cost interval are given. The results given for each set are based on only those functions on which the search completed.

The Time Constraint method is able to obtain a network for each function in all four of the function sets. However, the optimal cost intervals for this method are large since a lower bound on the cost is not obtained.

Little improvement is seen over this first method when the second method of lowering the cost bound by 90% is applied. In order to shrink the search space significantly enough to allow for a larger number of functions to be completed, too much of the search space is pruned from the smaller functions so that the cost intervals become too large. Little is gained from this method.

The final method using a parameterization to lower the cost bound is an effective method. Because the cost bound is lowered based on the size of the search, little additional pruning is performed for the smaller functions where an optimal function can be found. Thus this method can provide a better guarantee about the optimality of the networks produced. For the larger functions, frequent lowering of the cost bounds allows the search to be pruned significantly enough that the search can be completed for all but 3 of the functions. Enough of the search is completed to obtain a comparable cost network and since the search is completed a lower bound on the optimal cost can be given. This results in a much smaller interval for the optimal cost.

From this comparison, we obtain a near-optimal method for completing synthesis. We will combine the parameterized lowering method with the time constraint method. If the first method is not able to complete the search after 3 hours then the time constraint method will be used.

		Time Constraint	Lower Cost Bound ($p = 90\%$)	Parameterized Lowering ($c = 2, b = 500$)
Equiv	Completed	3980	3980	3980
	Avg. Search Space	5,069.92	53.17	1,442.07
	Avg. Network Cost	9.64	12.50	10.22
	Avg. Cost Interval	9.64	11.25	1.77
Class 1	Completed	24	24	24
	Avg. Search Space	7,612.13	32.08	974.83
	Avg. Network Cost	8.00	9.17	8.21
	Avg. Cost Interval	8.00	8.25	1.25
Class 2	Completed	12	10	11
	Avg. Search Space	232,259.25	1,003.20	20,608.73
	Avg. Network Cost	58.00	30.70	35.09
	Avg. Cost Interval	58.00	27.63	26.73
Bench	Completed	15	15	17
	Avg. Search Space	1,869,649.07	1,714.07	44,837.65
	Avg. Network Cost	50.07	48.67	49.94
	Avg. Cost Interval	50.07	43.80	38.71

Figure 6.6: Near-optimal Method Comparison

6.2.5.2 Additional Results

Using the chosen near-optimal method, a table of results for functions from both the function classes and the benchmark set of functions can be completed. These tables are given in Figure 6.7 and Figure 6.9. Those functions indicated with a * could only be completed using the Time Constraint method. Those without a star indicate the search was completed using the parameterized method. The optimal cost interval is the interval in which the optimal cost network must fall. The lower bound of this interval is the value of the cost bound when the algorithm completed the search. Since the search completed, all networks with

cost smaller than this cost bound have been searched and no network was found. Therefore the optimal network must have cost greater than this value. If the search is not completed, then this value must be 0 since the entire solution space was not searched. The upper bound of the interval is the value of the network found during the search. Any optimal cost network for the function must have cost at most equal to the cost of this network.

Function Class	Inputs	Outputs	Network Cost	Optimal Cost Interval	Time	Search Tree Size
AND	8	1	14	(6, 14]	3 s	4,239
AND	9	1	16	(6, 16]	5 s	5,402
NAND	8	1	13	(5, 13]	2 s	4,239
NAND	9	1	15	(5, 15]	4 s	5,402
OR	7	1	18	(10, 18]	4 s	4,902
OR	8	1	21	(9, 21]	6 s	6,672
OR	9	1	24	(10, 24]	7 s	8,740
NOR	7	1	19	(11, 19]	4 s	4,902
NOR	8	1	22	(10, 22]	5 s	6,672
NOR	9	1	25	(11, 25]	7 s	8,740
XOR	5	1	17	(9, 17]	4 s	4,956
XOR	6	1	21	(9, 21]	6 s	7,194
XNOR	5	1	40	(10, 40]	1 min	41,825
XNOR	6	1	111	(9, 111]	18 min	165,540
MAJ	6	1	36	(8, 36]	22 s	19,379
MAJ	7	1	73	(9, 73]	2 min	54,989
MAJ	8	1	118	(8, 118]	7 min	96,686
MUX	8	1	23	(9, 23]	7 s	12,325
MUX	9	1	28	(8, 28]	14 s	13,537
ADDER	5	3	34	(10, 34]	1 min	37,513
ADDER	6	4	76	(12, 76]	3 min	53,805
DECODER	4	16	90	(36, 90]	27 min	36,738

Figure 6.7: Near-optimal Results for Function Classes

Function Class	Inputs	Outputs	Network Cost	Optimal Cost Interval	Time	Search Tree Size
THRESH 1	7	1	18	(10, 18]	3 s	4,902
THRESH 1	8	1	21	(9, 21]	5 s	6,672
THRESH 1	9	1	24	(10, 24]	7 s	8,740
THRESH 2	5	1	25	(9, 25]	8 s	10,000
THRESH 2	6	1	31	(9, 31]	15 s	14,997
THRESH 2	7	1	34	(8, 34]	18 s	16,605
THRESH 2	8	1	39	(9, 39]	27 s	20,342
THRESH 2	9	1	85	(9, 85]	3 min	66,096
THRESH 3	5	1	18	(8, 18]	4 s	6,183
THRESH 3	6	1	50	(8, 50]	38 s	28,565
THRESH 3	7	1	68	(8, 68]	2 min	51,743
THRESH 3	8	1	105	(7, 105]	11 min	126,174
THRESH 3	9	1	209	(7, 209]	2 hrs	323,678
THRESH 4	6	1	36	(8, 36]	22 s	19,379
THRESH 4	7	1	73	(9, 73]	2 min	54,989
THRESH 4	8	1	158	(8, 158]	34 min	209,536
THRESH 4	9					
THRESH 5	6	1	21	(7, 21]	5 s	8,030
THRESH 5	7	1	47	(7, 47]	43 s	29,629
THRESH 5	8	1	118	(8, 118]	7 min	96,686
THRESH 5	9					
THRESH 6	7	1	36	(8, 36]	15 s	18,040
THRESH 6	8	1	84	(8, 84]	2 min	57,669
THRESH 6	9	1	195	(7, 195]	55 min	221,570
THRESH 7	8	1	49	(7, 49]	29 s	26,526
THRESH 7	9	1	144	(6, 144]	6 min	104,039
THRESH 8	8	1	49	(7, 49]	29 s	26,526
THRESH 8	9	1	144	(6, 144]	6 min	104,039
THRESH 9	9	1	16	(6, 16]	4 s	5,402

Figure 6.8: Near-optimal Results for Threshhold Class

Name	Inputs	Outputs	Network Cost	Optimal Cost Interval	Time	Search Tree Size
2bit_adder	5	3	34	(10, 34]	1 min	37513
b1	3	3	10	(10, 10]	0 s	450
C17	5	2	6	(6, 6]	0 s	23
cm138a	6	8	59	(13, 59]	2 min	46108
cm152a	11	1	57	(5, 57]	2 min	42735
cm42a	4	10	35	(15, 35]	29 s	13796
cm82a	4	3	34	(12, 34]	1 min	32276
cmb	16	4	121	(7, 121]	2 hrs	146648
decod	5	16	131	(35, 131]	21 min	91384
majority	5	1	11	(9, 11]	0 s	1459
muroga	3	1	7	(7, 7]	0 s	105
oai22	4	1	7	(7, 7]	0 s	87
partialMux	6	1	12	(10, 12]	1 s	1913
small	3	1	2	(2, 2]	0 s	7
tcon	17	8	25	(19, 25]	33 min	4186
x2	10	7	103	(13, 103]	7 min	84235
z4ml	7	4	195	(11, 195]	2 hrs	259315

Figure 6.9: Near-optimal Results for Benchmarks

Relaxing the optimality constraint has allowed us to complete the search for functions with as many as 17 inputs and 16 outputs. Networks as large as 209 gates were completed. The minimum cost interval contains 2 gates while the maximum contains 202 and the average contains 45 gates.

6.3 Comparison with Established Heuristic Method

By combining the optimal results presented in Chapter 4 with the larger near-optimal results presented here, enough data have been compiled so that an evaluation of a heuristic multi-level synthesis methods can be performed. As an example of this evaluation, we have chosen to evaluate the well-known synthesis package ABC [ABC 07].

An evaluation of the networks produced by ABC compares the distance between the network produced by ABC and the optimal cost for the function. For those functions where optimal networks are known, we will be able to compare the networks produced by ABC to these optimal networks. For those larger networks where only near-optimal results are known, we can determine if the network produced by ABC falls within the range determined by the near-optimal results and compare the cost of the network produced by ABC to the near-optimal network found by the BESS.

The networks used for this comparison were created using the {NAND2} building block set and the cost function which counts the number of gates in the network. Thus the library we will use for ABC will contain a primary input port, a primary output port, and a 2-input, 1-output NAND gate. We will have ABC use the script resyn2 and map for area optimization.

Figure 6.10 provides a comparison of the results obtained using ABC to those found using BESS. Graph (a) gives the results for functions where an optimal network has been found; graph (b) compares the

algorithms on functions where only an optimal cost interval can be given. In both graphs, each function is represented as a position on the x-axis. The costs of the networks are plotted on the y-axis. The results found by ABC are the points given in gray. In graph (a), the black points indicate the optimal cost of the network. In graph (b), the points indicated with + give the upper bound of the cost interval, while the points denoted with — give the lower bound for the cost interval.

ABC was able to produce an optimal network for 340 out of 4,043 functions where the optimal network is known. For those functions where ABC did not produce an optimal network it was an average of 36% larger than the optimal network, with the largest network requiring 14 additional gates.

For those functions where only an interval for the optimal solution is known, ABC produced a network that was within this interval for 52 out of the 60 functions. Of these functions 15 had the same cost as the upper bound on the interval (the cost of the network found by the near-optimal version of BESS), and 37 had cost smaller than the upper bound. For those 8 functions on which ABC produced a network with cost outside of the optimal cost interval, the cost of a network was on average 19% larger than the network found by BESS, with the largest using only 10 additional gates.

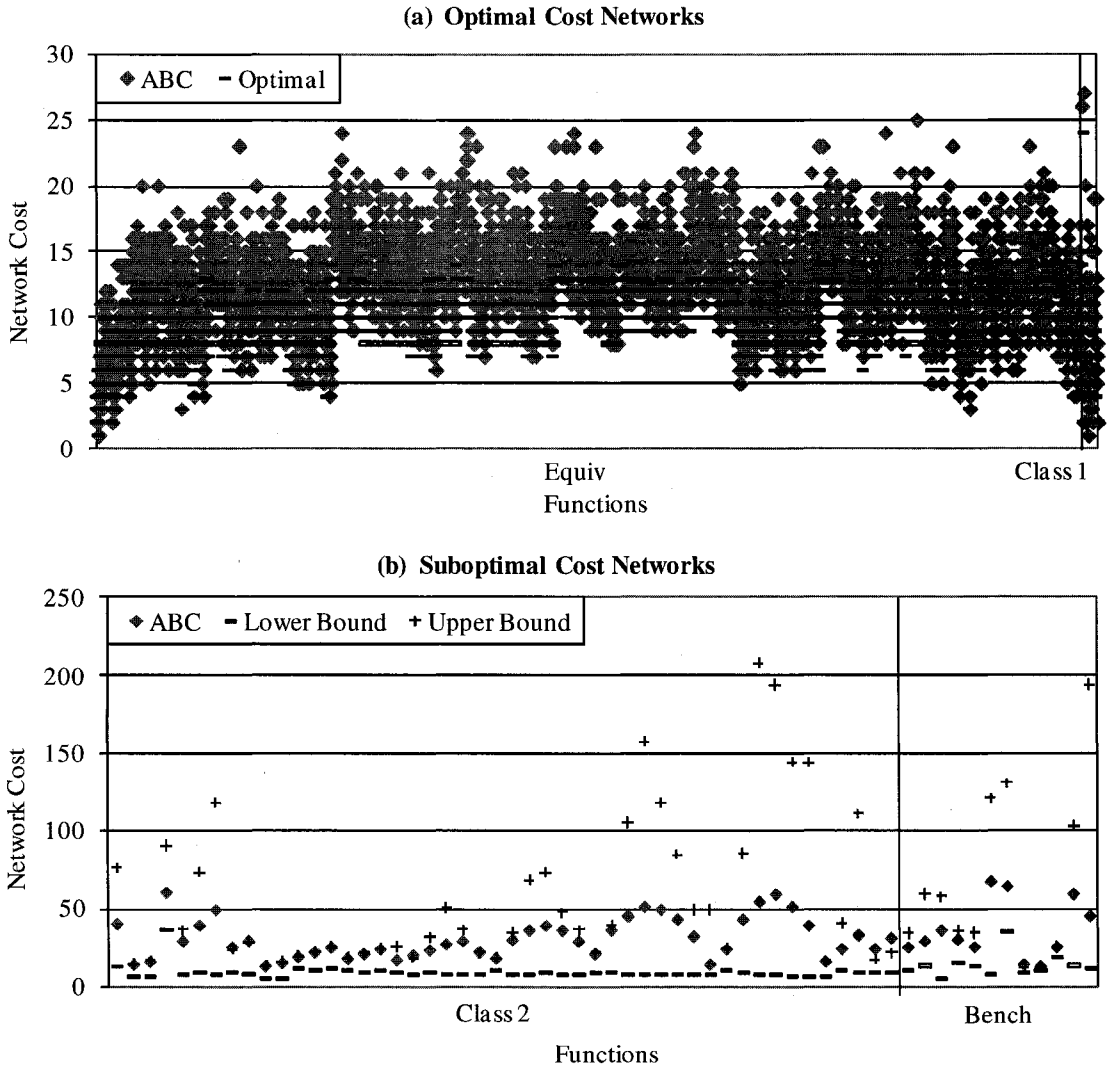


Figure 6.10: Results of ABC on Optimal Networks

This comparison shows that ABC performs well for a majority of functions. Since ABC was able to produce networks within the cost interval for the majority of the larger functions, we can conclude BESS can provide a way to evaluate the networks produced by heuristic methods for multi-level synthesis but cannot act as a robust heuristic for performing synthesis on any problem.

6.4 Summary

The exact synthesis algorithm, BESS, described in this work has a structure that lends itself well to producing near-optimal results. The branch-and-bound nature of the algorithm allows complete networks to be found quickly and then improvements upon the cost of network made as the search continues. In this chapter we explored ways that we can exploit this property of BESS to provide near-optimal results.

First we investigated four variations on the near-optimal algorithm providing reasons to consider each variation and results of this variation on smaller functions where the optimal cost is known and larger functions where it is not. By comparing the methods we were able to determine that a combination of the parameterization method and the Time Constraint method was the best choice. We are able to complete synthesis for all of the functions which did not complete earlier and an interval for the optimal cost was given for these functions.

We concluded this chapter by using both the optimal and near-optimal results of BESS to evaluate a heuristic synthesis method, ABC. We were able to determine that for 10% of the functions attempted, ABC was able to complete either an optimal network or a network contained within the optimal cost interval. For the remaining functions, the networks produced by ABC were an average of 36% larger than either the optimal cost or the optimal cost interval.

Chapter 7

Conclusions

7.1 Future Directions and Applications

There are several directions in which the investigation presented here can be taken. Modifications to BESS can be made to allow for different versions of synthesis similar to those described in Chapter 5. Additional techniques can also be investigated to improve the search of BESS for specialized synthesis options or specific types of functions. Finally, the results of our exact synthesis method can be integrated into existing heuristic methods.

7.1.1 Further Modification of the Algorithm

The flexibility of BESS allows for the design of optimal networks under arbitrary restrictions on the networks. Thus, additional modifications to the algorithm similar to those presented in Chapters 5 and 6 can be made. Synthesis with new cost functions and building blocks can be performed with these modifications. We saw several cost functions in Chapter 5 that could be used. These cost functions can be extended to any function based on a property of the network including the traditional properties of area, critical path delay, and power consumption or alternate properties such as the number of literals in the factored form. Combinations of these cost functions with fan-in, fan-out, and level restrictions can be performed using the techniques outlined in Chapter 5.

Alternate building block sets can also be employed. Following the process outlined in Chapter 5, the relationship between the node's input and output functions, the functional implications required to maintain this relationship, and the functional consistency constraint can be determined based on the chosen building blocks.

Building blocks with multiple outputs could also be considered. For example, the NAND-AND gate could be used as a building block. In this case, the node data structure of the algorithm must keep track of the minterm covering for each output of the gate. The connectible properties of the gate must also incorporate both outputs. The covering details for the individual functions will remain the same, however.

Finally, modifications for specific building block sets and cost functions can be made to the algorithm to optimize the search on this particular specification of the synthesis problem. While many of the algorithmic techniques added to the algorithm in this thesis are helpful no matter what synthesis problem

was solved, some were specifically tuned to the use of NAND2 gates and optimization under the number of gates. In the future, lower bounding techniques on the number of gates can be used in cases when structural implications are not possible or when the number of interconnections or number of levels are incorporated in the cost function.

7.1.2 Parallelization & Random Restarts

The algorithm presented here lends itself easily to parallelization. At the beginning of the search, the alternate methods of covering the first several selected minterms can be given to separate processors. The only data that need to be shared among the processes are the cost of the current smallest network. When a possible solution network is found this must be broadcast to all of the other processors so that the search can be pruned accordingly.

The obvious benefit of performing the search in parallel is that less time will be required since the search is split among multiple processors. Additionally, by splitting the search in this way, the amount of the solution space that has to be searched may shrink as well. The original algorithm searches the space in a depth-first manner. Thus, if a poor choice is made at the beginning of the search it may be the case that a low cost network is not found until a significant portion of the search has been completed. However, by distributing the search among several processors at the beginning of the search, some breadth is added to this depth-first search. Therefore the low cost network can be found by one processor and the cost shared with the others. This will prune the search significantly for those cases when, while running on a single processor, the optimal network is not found until later in the search and will do no harm for those cases when the optimal network is the initial network found.

The utilization of random restarts of the algorithm could also help to improve the run-time of the algorithm. As in parallelization, if a poor choice is made at the beginning of the search then a restart of the algorithm will allow some breadth to be added into the search. It is possible that a low-cost network can be found sooner by exploring a new branch early in the search. The disadvantage of using the restart is that the entire search must still be completed following the restart. Thus, portions of the search space must be repeated.

7.1.3 Conflicts and Constraint Learning

Chapter 3 presented several situations that may result in a conflict during the search. Currently the algorithm simply backtracks from this conflict and proceeds with the search. However, each conflict presents an opportunity for learning. Similar to constraint learning employed in satisfiablity algorithms, information about the network when a conflict occurs can be used to learn which covering sequences result in this type of conflict. The savings in the size of the search will be evident as portions of the search tree which lead to such conflicts are removed.

The difference between the search performed here and a typical backtracking search makes the process of learning more difficult. In this search, the set of decision variables is constantly changing. This makes it

difficult to analyze the decisions which led to the conflict and how to represent the learned constraint in each new network.

7.1.4 Heuristic Methods

As we briefly described in Chapter 6, methods for exact synthesis can be used within the broader scope of logic synthesis. We described how the results of BESS can be used to evaluate existing heuristic methods for synthesis by comparing both optimal and near-optimal networks to those produced by the heuristic method. Additional research can be employed in this area. First, the set of functions on which heuristic methods are evaluated can be expanded. Larger functions and networks can be created by combining two or more optimal networks into one larger network. These new test functions will have a known upper bound for the optimal cost of the network.

The optimal results given here can also be used to synthesize larger functions. There are two ways that this can be accomplished. Given an initial network for a function, the network can be partitioned into smaller more manageable pieces, and then these subnetworks optimized using the exact synthesis algorithm. A second method would be to use the optimal networks produced by the algorithm as the building blocks within a heuristic method. The networks produced by either of these methods will be a reasonable although not necessarily optimal implementation of the function being composed of optimal subnetworks.

Finally, BESS can be used as a pre-processing technique for more robust heuristic methods. As Chapter 6 showed, the initial network found by the algorithm was very often close to (if not exactly) the optimal network. Therefore this initial network found by the algorithm can be used to produce near-optimal initial partial networks for a set of functions which can then be given to such a heuristic method for further improvements.

7.1.5 Catalog of Optimal Networks

The database of optimal networks provided here can be extended providing more information about the structures of optimal circuits. The completion of optimal networks for all 5-input functions was limited in this work due to the total number of possible functions. While further progress could be made in completing this database, a more useful direction for future work would be the construction of a catalog of optimal networks for a selection of "interesting" functions. Similar to those results presented in Section 4.3.2 a catalog of common functions seen often in synthesis would be useful to logic designers when completing synthesis for more complex systems. This catalog should contain optimal networks for these functions under a variety of synthesis constraints.

7.2 Summary of Thesis

In this thesis we provided a theoretical and experimental investigation of exact synthesis. We began by providing a general formulation of optimal synthesis as a dynamic search problem and described a branch-and-bound search algorithm, BESS, that can be used for completing optimal synthesis based on this formulation.

By formulating optimal synthesis as a search problem, we were able to analyze the solution space that is searched by the algorithm and precisely define the decision variables which drive the search. As part of this analysis we provided a proof of completeness which showed that the algorithm will explore all possible implementations of a Boolean function guaranteeing that the optimal implementation will be found. We also gave a proof of convergence. In this proof, we discussed the three requirements needed for the initial path of the search to guarantee that an initial complete network would be found and that the search would complete. Finally, we performed a worst-case analysis of the search space using conventional height and width analysis of the search tree. This analysis showed a greater than double exponential size for the search tree based on the number of inputs to the function.

From this formulation we constructed a branch-and-bound search algorithm, BESS, to complete optimal synthesis. Through our description of BESS we demonstrated its three key features. The first is the flexibility of the algorithm. The algorithm can easily handle a variety of synthesis options as we demonstrated in Chapter 5. The second is its completeness: given enough resources, the algorithm will produce an optimal implementation for a given Boolean function. Finally, in Chapter 6, we showed that the algorithm provides for the possibility of relaxing the optimality constraint so that complete networks for large functions can be produced with near-optimal cost.

Since the problem of optimal synthesis is known to be combinatorially difficult, when constructing BESS we performed an extensive study of various search strategies which helped to provide a more efficient search. In Chapters 3 and 4 we gave both the theoretical and empirical arguments to support the use of these strategies including decisions heuristics, decision implications, and pruning rules.

In the second half of this thesis we discussed the experimental results we obtained using BESS. First, we constructed a database of optimal networks for over 8,000 functions with five or fewer variables. These networks were created using 2-input NAND gates and were optimized over the number of gates in the network. We completed the database by optimally synthesizing one representative function for each P-equivalence class. This database confirmed the optimal networks previously known for all 2-, 3-, and all but one 4-input functions and provided new optimal results for the last 4-input function and 4,745 5-input functions.

We also used BESS to construct networks for larger functions. First, by analyzing the structures of the optimal networks on a variety of common functions we provided optimal implementations and cost formula for the n-input functions from these classes. For four of these classes (AND, NAND, OR, and NOR) we proved

the optimality of the results. Second, by relaxing the optimality constraint we used BESS to construct larger networks which are close to optimal. The branch-and-bound nature of the algorithm allowed us to easily modify the bounding constraint to produce networks which are a known distance from optimal. Using this technique we completed near-optimal networks for a selection of functions from the MCNC benchmark suite.

Finally, we investigated one way in which the results provided here can be used within the broader scope of logic synthesis. Using both the optimal and near-optimal results, we evaluated the heuristic synthesis system ABC. This evaluation determined that ABC produced optimal results for 10% functions attempted, and for the remaining functions, the networks produced by ABC were an average of 36% larger than either the optimal cost or the optimal cost interval provided by BESS

Appendix

A.1 Representative Sets

The following three tables are representative functions selected from the P- and NPN-equivalence classes for functions with 2, 3, and 4 inputs. Each functions is described by an integer representation, binary representation, and Boolean function. The binary representation is obtained from the truth table for the function with the most significant digit representing the truth table line $x_1'x_2'$ and the least significant digit representing the truth table line x_1x_2.

For example the function $x_1 \vee x_2$ has the truth table:

x_1	0	0	1	1
x_2	0	1	0	1
$x_1 \vee x_2$	0	1	1	1

Therefore the binary representation of $x_1 \vee x_2$ is 0111 and the integer representation is 7.

2 input P representative set

Integer Representation	Binary Representation	Boolean Function
1	0001	ab
2	0010	ab'
6	0110	ab' + a'b
7	0111	a + b
8	1000	a'b'
9	1001	ab + a'b'
11	1011	a + b'
14	1110	a' + b'

3 input P representative set

Integer Representation	Binary Representation	Boolean Function
1	00000001	abc
2	00000010	abc'
6	00000110	abc' + ab'c
7	00000111	ab + ac
8	00001000	ab'c'
9	00001001	abc + ab'c'
11	00001011	ab + ac'
14	00001110	ac' + ab'
22	00010110	abc' + ab'c + a'bc
23	00010111	ab + ac + bc
24	00011000	ab'c' + a'bc
25	00011001	bc + ab'c'
26	00011010	ac' + a'bc
27	00011011	ab + ac' + bc
30	00011110	ac' + ab' + a'bc
31	00011111	a + bc
40	00101000	ab'c' + a'bc'
41	00101001	abc + ab'c' + a'bc'
42	00101010	ac' + bc'
43	00101011	ab + ac' + bc'
44	00101100	ab' + a'bc'
45	00101101	ac + ab' + a'bc'
46	00101110	ac' + ab' + bc'
47	00101111	a + bc'
61	00111101	ac + ab' + bc + a'b
62	00111110	ac' + ab' + bc' + a'b
104	01101000	ab'c' + a'bc' + a'b'c
105	01101001	abc + ab'c' + a'bc' + a'b'c
106	01101010	ac' + bc' + a'b'c
107	01101011	ab + ac' + bc' + a'b'c
110	01101110	ac' + ab' + bc' + b'c
111	01101111	a + bc' + b'c
126	01111110	ac' + ab' + bc' + a'b + b'c + a'c
127	01111111	a + b + c
128	10000000	a'b'c'

Integer Representation	Binary Representation	Boolean Function
129	10000001	abc + a'b'c'
130	10000010	abc' + a'b'c'
131	10000011	ab + a'b'c'
134	10000110	abc' + ab'c + a'b'c'
135	10000111	ab + ac + a'b'c'
137	10001001	b'c' + abc
138	10001010	ac' + b'c'
139	10001011	ab + ac' + b'c'
142	10001110	ac' + ab' + b'c'
143	10001111	a + b'c'
150	10010110	abc' + ab'c + a'bc + a'b'c'
151	10010111	ab + ac + bc + a'b'c'
152	10011000	b'c' + a'bc
154	10011010	ac' + b'c' + a'bc
155	10011011	ab + ac' + bc + b'c'
158	10011110	ac' + ab' + b'c' + a'bc
159	10011111	a + bc + b'c'
168	10101000	b'c' + a'c'
169	10101001	b'c' + a'c' + abc
171	10101011	c' + ab
172	10101100	ab' + b'c' + a'c'
173	10101101	ac + ab' + b'c' + a'c'
174	10101110	c' + ab'
188	10111100	ab' + a'b + b'c' + a'c'
189	10111101	ac + ab' + bc + a'b + b'c' + a'c'
190	10111110	c' + ab' + a'b
191	10111111	a + b + c'
232	11101000	b'c' + a'c' + a'b'
233	11101001	b'c' + a'c' + a'b' + abc
234	11101010	c' + a'b'
235	11101011	c' + ab + a'b'
239	11101111	a + c' + b'
254	11111110	c' + b' + a'

4 input NPN representative set

Integer Representation	Binary Representation	Boolean Function
1	0000000000000001	abcd
6	0000000000000110	abcd' + abc'd
7	0000000000000111	abc + abd
22	0000000000010110	abcd' + abc'd + ab'cd
23	0000000000010111	abc + abd + acd
24	0000000000011000	abc'd' + ab'cd
25	0000000000011001	acd + abc'd'
27	0000000000011011	abc + abd' + acd
30	0000000000011110	abd' + abc' + ab'cd
31	0000000000011111	ab + acd
61	0000000000111101	abd + abc' + acd + ab'c
105	0000000001101001	abcd + abc'd' + ab'cd' + ab'c'd
107	0000000001101011	abc + abd' + acd' + ab'c'd
111	0000000001101111	ab + acd' + ac'd
126	0000000001111110	abd' + abc' + acd' + ab'c + ac'd + ab'd
127	0000000001111111	ab + ac + ad
278	0000000100010110	abcd' + abc'd + ab'cd + a'bcd
279	0000000100010111	abc + abd + acd + bcd
280	0000000100011000	abc'd' + ab'cd + a'bcd
281	0000000100011001	acd + bcd + abc'd'
282	0000000100011010	abd' + ab'cd + a'bcd
283	0000000100011011	abc + abd' + acd + bcd
286	0000000100011110	abd' + abc' + ab'cd + a'bcd
287	0000000100011111	ab + acd + bcd
300	0000000100101100	abc' + ab'cd' + a'bcd
301	0000000100101101	abd + abc' + bcd + ab'cd'
303	0000000100101111	ab + acd' + bcd
316	0000000100111100	abc' + ab'c + a'bcd
317	0000000100111101	abd + abc' + acd + ab'c + bcd
318	0000000100111110	abd' + abc' + acd' + ab'c + a'bcd
319	0000000100111111	ab + ac + bcd
360	0000000101101000	abc'd' + ab'cd' + ab'c'd + a'bcd
361	0000000101101001	bcd + abc'd' + ab'cd' + ab'c'd
362	0000000101101010	abd' + acd' + ab'c'd + a'bcd
363	0000000101101011	abc + abd' + acd' + bcd + ab'c'd
366	0000000101101110	abd' + abc' + acd' + ac'd + a'bcd
367	0000000101101111	ab + acd' + ac'd + bcd
382	0000000101111110	abd' + abc' + acd' + ab'c + ac'd + ab'd + a'bcd
383	0000000101111111	ab + ac + ad + bcd

Integer Representation	Binary Representation	Boolean Function
384	0000000110000000	ab'c'd' + a'bcd
385	0000000110000001	bcd + ab'c'd'
386	0000000110000010	abcd' + ab'c'd' + a'bcd
387	0000000110000011	abc + bcd + ab'c'd'
390	0000000110000110	abcd' + abc'd + ab'c'd' + a'bcd
391	0000000110000111	abc + abd + bcd + ab'c'd'
393	0000000110001001	ac'd' + bcd
395	0000000110001011	abc + abd' + ac'd' + bcd
399	0000000110001111	ab + ac'd' + bcd
406	0000000110010110	abcd' + abc'd + ab'cd + ab'c'd' + a'bcd
407	0000000110010111	abc + abd + acd + bcd + ab'c'd'
408	0000000110011000	ac'd' + ab'cd + a'bcd
409	0000000110011001	acd + ac'd' + bcd
410	0000000110011010	abd' + ac'd' + ab'cd + a'bcd
411	0000000110011011	abc + abd' + acd + ac'd' + bcd
414	0000000110011110	abd' + abc' + ac'd' + ab'cd + a'bcd
415	0000000110011111	ab + acd + ac'd' + bcd
424	0000000110101000	ac'd' + ab'd' + a'bcd
425	0000000110101001	ac'd' + ab'd' + bcd
426	0000000110101010	ad' + a'bcd
427	0000000110101011	ad' + abc + bcd
428	0000000110101100	abc' + ac'd' + ab'd' + a'bcd
429	0000000110101101	abd + abc' + ac'd' + ab'd' + bcd
430	0000000110101110	ad' + abc' + a'bcd
431	0000000110101111	ab + ad' + bcd
444	0000000110111100	abc' + ab'c + ac'd' + ab'd' + a'bcd
445	0000000110111101	abd + abc' + acd + ab'c + ac'd' + ab'd' + bcd
446	0000000110111110	ad' + abc' + ab'c + a'bcd
447	0000000110111111	ab + ac + ad' + bcd
488	0000000111101000	ac'd' + ab'd' + ab'c' + a'bcd
489	0000000111101001	ac'd' + ab'd' + ab'c' + bcd
490	0000000111101010	ad' + ab'c' + a'bcd
491	0000000111101011	ad' + abc + ab'c' + bcd
494	0000000111101110	ad' + ac' + a'bcd
495	0000000111101111	ab + ad' + ac' + bcd
510	0000000111111110	ad' + ac' + ab' + a'bcd
829	0000001100111101	abd + abc' + acd + ab'c + bcd + a'bc
854	0000001101010110	ac'd + ab'd + bcd' + a'bc
855	0000001101010111	ad + bc

Integer Representation	Binary Representation	Boolean Function
856	0000001101011000	ab'd + a'bc + abc'd'
857	0000001101011001	acd + ab'd + bcd + a'bc + abc'd'
858	0000001101011010	abd' + ab'd + bcd' + a'bc
859	0000001101011011	bc + abd' + acd + ab'd
862	0000001101011110	abd' + abc' + ac'd + ab'd + bcd' + a'bc
863	0000001101011111	ab + ad + bc
872	0000001101101000	a'bc + abc'd' + ab'cd' + ab'c'd
873	0000001101101001	bcd + a'bc + abc'd' + ab'cd' + ab'c'd
874	0000001101101010	abd' + acd' + bcd' + a'bc + ab'c'd
875	0000001101101011	bc + abd' + acd' + ab'c'd
876	0000001101101100	abc' + ac'd + a'bc + ab'cd'
877	0000001101101101	abd + abc' + ac'd + bcd + a'bc + ab'cd'
878	0000001101101110	abd' + abc' + acd' + ac'd + bcd' + a'bc
879	0000001101101111	ab + bc + acd' + ac'd
892	0000001101111100	abc' + ab'c + ac'd + ab'd + a'bc
893	0000001101111101	ad + abc' + ab'c + bcd + a'bc

Integer Representation	Binary Representation	Boolean Function
894	0000001101111110	abd' + abc' + acd' + ab'c + ac'd + ab'd + bcd' + a'bc
961	0000001111000001	ab'c' + bcd + a'bc
965	0000001111000101	abd + ac'd + ab'c' + bcd + a'bc
966	0000001111000110	ac'd + ab'c' + bcd' + a'bc
967	0000001111000111	bc + abd + ac'd + ab'c'
980	0000001111010100	ac'd + ab'd + ab'c' + a'bc
981	0000001111010101	ad + ab'c' + bcd + a'bc
982	0000001111010110	ac'd + ab'd + ab'c' + bcd' + a'bc
983	0000001111010111	ad + bc + ab'c'
984	0000001111011000	ab'd + ac'd' + ab'c' + a'bc
985	0000001111011001	acd + ab'd + ac'd' + ab'c' + bcd + a'bc
987	0000001111011011	bc + abd' + acd + ab'd + ac'd' + ab'c'
988	0000001111011100	ac' + ab'd + a'bc
989	0000001111011101	ad + ac' + bcd + a'bc
990	0000001111011110	ac' + abd' + ab'd + bcd' + a'bc

A.2 Function Classes

The functions given here represent common Boolean functions. Each set will contain Boolean functions which evaluate the logic operation on an increasing number of inputs. The ten function classes are: AND, NAND, OR, NOR, XOR, XNOR, MAJORITY, MULTIPLEXER, ADDER, and DECODER.

AND

Input Variables	Boolean Function
2	ab
3	abc
4	abcd
5	abcde
6	abcdef
7	abcdefg
8	abcdefgh
9	abcdefghi

NAND

Input Variables	Boolean Function
2	a' + b'
3	a' + b' + c'
4	a' + b' + c' + d'
5	a' + b' + c' + d' + e'
6	a' + b' + c' + d' + e' + f'
7	a' + b' + c' + d' + e' + f' + g'
8	a' + b' + c' + d' + e' + f' + g' + h'
9	a' + b' + c' + d' + e' + f' + g' + h' + i'

OR

Input Variables	Boolean Function
2	a + b
3	a + b + c
4	a + b + c + d
5	a + b + c + d + e
6	a + b + c + d + e + f
7	a + b + c + d + e + f + g
8	a + b + c + d + e + f + g + h
9	a + b + c + d + e + f + g + h + i

NOR

Input Variables	Boolean Function
2	a'b'
3	a'b'c'
4	a'b'c'd'
5	a'b'c'd'e'
6	a'b'c'd'e'f'
7	a'b'c'd'e'f'g'
8	a'b'c'd'e'f'g'h'
9	a'b'c'd'e'f'g'h'i'

XOR

Input Variables	Boolean Function
2	a ⊕ b
3	a ⊕ b ⊕ c
4	a ⊕ b ⊕ c ⊕ d
5	a ⊕ b ⊕ c ⊕ d ⊕ e
6	a ⊕ b ⊕ c ⊕ d ⊕ e ⊕ f
7	a ⊕ b ⊕ c ⊕ d ⊕ e ⊕ f ⊕ g
8	a ⊕ b ⊕ c ⊕ d ⊕ e ⊕ f ⊕ g ⊕ h
9	a ⊕ b ⊕ c ⊕ d ⊕ e ⊕ f ⊕ g ⊕ h ⊕ i

XNOR

Input Variables	Boolean Function
2	(a ⊕ b)′
3	(a ⊕ b ⊕ c)′
4	(a ⊕ b ⊕ c ⊕ d)′
5	(a ⊕ b ⊕ c ⊕ d ⊕ e)′
6	(a ⊕ b ⊕ c ⊕ d ⊕ e ⊕ f)′
7	(a ⊕ b ⊕ c ⊕ d ⊕ e ⊕ f ⊕ g)′
8	(a ⊕ b ⊕ c ⊕ d ⊕ e ⊕ f ⊕ g ⊕ h)′
9	(a ⊕ b ⊕ c ⊕ d ⊕ e ⊕ f ⊕ g ⊕ h ⊕ i)′

MAJORITY

Input Variables	Boolean Function
2	ab
3	a (b + c) + bc
4	a (b (c + d) + cd)+ bcd
5	a (b (c + d + e) + c (d + e) + de) + b (c (d + e) + de) + cde
6	a (b (c (d + e + f) + d (e + f) + ef) + c (d (e + f) + ef) + def) + b (c (d (e + f) + ef) + def) + cdef

MULTIPLEXER (MUX)

Input Variables	Boolean Function
3	$ac + a'b$
5	$a'b'c + a'bd + ab'e$
5	$a'b'c + a'bd + abe$
5	$a'bc + ab'd + abe$
5	$a'bc + ab'd + abe$
6	$a'b'c + a'bd + ab'e + abf$
8	$a'bc'd + abc'e + ab'c'f + ab'cg + abch$
9	$a'bcd + a'bc'e + abc'f + ab'c'g + ab'ch + abci$

ADDER

Description	Input Variables	Boolean Functions
Half	2	$S = a \oplus b$ $C = ab$
Full	3	$S = a \oplus b \oplus c$ $C = ab + ac + bc$
2 bit	4	$S_0 = a_0 \oplus b_0$ $S_1 = a_1 \oplus b_1 \oplus (a_0b_0)$ $C_1 = a_1b_1a_0b_0$
3 bit	6	$S_0 = a_0 \oplus b_0$ $S_1 = a_1 \oplus b_1 \oplus (a_0b_0)$ $S_2 = a_2 \oplus b_2 \oplus (a_1b_1a_0b_0)$ $C_2 = a_2b_2a_1b_1a_0b_0$

DECODER

Input Variables	Boolean Functions
2	$f_1 = a'b'$ $f_2 = a'b$ $f_3 = ab'$ $f_4 = ab$
3	$f_1 = a'b'c'$ $f_2 = a'b'c$ $f_3 = a'bc'$ $f_4 = a'bc$ $f_5 = ab'c'$ $f_6 = ab'c$ $f_7 = abc'$ $f_8 = abc$
4	$f_1 = a'b'c'd'$ $f_2 = a'b'c'd$ $f_3 = a'b'cd'$ $f_4 = a'b'cd$ $f_5 = a'bc'd'$ $f_6 = a'bc'd$ $f_7 = a'bcd'$ $f_8 = a'bcd$ $f_9 = ab'c'd'$ $f_{10} = ab'c'd$ $f_{11} = ab'cd'$ $f_{12} = ab'cd$ $f_{13} = abc'd'$ $f_{14} = abc'd$ $f_{15} = abcd'$ $f_{16} = abcd$

THRESHOLD

Input Variables	Threshold Level	Boolean Function
2	1	$a + b$
3	1	$a + b + c$
4	1	$a + b + c + d$
5	1	$a + b + c + d + e$
6	1	$a + b + c + d + e + f$
7	1	$a + b + c + d + e + f + g$
8	1	$a + b + c + d + e + f + g + h$
9	1	$a + b + c + d + e + f + g + h + i$
2	2	ab
3	2	$ab + ac + bc$
4	2	$ab + ac + ad + bc + bd + cd$
5	2	$ab + ac + ad + ae + bc + bd + be + cd + ce + de$
6	2	$ab + ac + ad + ae + af + bc + bd + be + bf + cd + ce + cf + de + df + ef$
7	2	$ab + ac + ad + ae + af + ag + bc + bd + be + bf + bg + cd + ce + cf + cg + de + df + dg + ef + eg + fg$
8	2	$ab + ac + ad + ae + af + ag + ah + bc + bd + be + bf + bg + bh + cd + ce + cf + cg + ch + de + df + dg + dh + ef + eg + eh + fg + fh + gh$
9	2	$ab + ac + ad + ae + af + ag + ah + ai + bc + bd + be + bf + bg + bh + bi + cd + ce + cf + cg + ch + ci + de + df + dg + dh + di + ef + eg + eh + ei + fg + fh + fi + gh + gi + hi$
3	3	abc

Input Variables	Threshold Level	Boolean Function
4	3	abc + abd + acd + bcd
5	3	abc + abd + abe + acd + ace + ade + bcd + bce + bde + cde
6	3	abc + abd + abe + abf + acd + ace + acf + ade + adf + aef + bcd + bce + bcf + bde + bdf + bef + cde + cdf + cef + def
7	3	abc + abd + abe + abf + abg + acd + ace + acf + acg + ade + adf + adg + aef + aeg + afg + bcd + bce + bcf + bcg + bde + bdf + bdg + bef + beg + bfg + cde + cdf + cdg + cef + ceg + cfg + def + deg + dfg + efg
8	3	abc + abd + abe + abf + abg + abh + acd + ace + acf + acg + ach + ade + adf + adg + adh + aef + aeg + aeh + afg + afh + agh + bcd + bce + bcf + bcg + bch + bde + bdf + bdg + bdh + bef + beg + beh + bfg + bfh + bgh + cde + cdf + cdg + cdh + cef + ceg + ceh + cfg + cfh + cgh + def + deg + deh + dfg + dfh + dgh + efg + efh + egh + fgh
9	3	abc + abd + abe + abf + abg + abh + abi + acd + ace + acf + acg + ach + aci + ade + adf + adg + adh + adi + aef + aeg + aeh + aei + afg + afh + afi + agh + agi + ahi + bcd + bce + bcf + bcg + bch + bci + bde + bdf + bdg + bdh + bdi + bef + beg + beh + bei + bfg + bfh + bfi + bgh + bgi + bhi + cde + cdf + cdg + cdh + cdi + cef + ceg + ceh + cei + cfg + cfh + cfi + cgh + cgi + chi + def + deg + deh + dei + dfg + dfh + dfi + dgh + dgi + dhi + efg + efh + efi + egh + egi + ehi + fgh + fgi + fhi + ghi
4	4	abcd
5	4	abcd + abce + abde + acde + bcde
6	4	abcd + abce + abcf + abde + abdf + abef + acde + acdf + acef + adef + bcde + bcdf + bcef + bdef + cdef
7	4	abcd + abce + abcf + abcg + abde + abdf + abdg + abef + abeg + abfg + acde + acdf + acdg + acef + aceg + acfg + adef + adeg + adfg + aefg + bcde + bcdf + bcdg + bcef + bceg + bcfg + bdef + bdeg + bdfg + befg + cdef + cdeg + cdfg + cefg + defg
8	4	abcd + abce + abcf + abcg + abch + abde + abdf + abdg + abdh + abef + abeg + abeh + abfg + abfh + abgh + acde + acdf + acdg + acdh + acef + aceg + aceh + acfg + acfh + acgh + adef + adeg + adeh + adfg + adfh + adgh + aefg + aefh + aegh + afgh + bcde + bcdf + bcdg + bcdh + bcef + bceg + bceh + bcfg + bcfh + bcgh + bdef + bdeg + bdeh + bdfg + bdfh + bdgh + befg + befh + begh + bfgh + cdef + cdeg + cdeh + cdfg + cdfh + cdgh + cefg + cefh + cegh + cfgh + defg + defh + degh + dfgh + efgh
9	4	abcd + abce + abcf + abcg + abch + abci + abde + abdf + abdg + abdh + abdi + abef + abeg + abeh + abei + abfg + abfh + abfi + abgh + abgi + abhi + acde + acdf + acdg + acdh + acdi + acef + aceg + aceh + acei + acfg + acfh + acfi + acgh + acgi + achi + adef + adeg + adeh + adei + adfg + adfh + adfi + adgh + adgi + adhi + aefg + aefh + aefi + aegh + aegi + aehi + afgh + afgi + afhi + aghi + bcde + bcdf + bcdg + bcdh + bcdi + bcef + bceg + bceh + bcei + bcfg + bcfh + bcfi + bcgh + bcgi + bchi + bdef + bdeg + bdeh + bdei + bdfg + bdfh + bdfi + bdgh + bdgi + bdhi + befg + befh + befi + begh + begi + behi + bfgh + bfgi + bfhi + bghi + cdef + cdeg + cdeh + cdei + cdfg + cdfh + cdfi + cdgh + cdgi + cdhi + cefg + cefh + cefi + cegh + cegi + cehi + cfgh + cfgi + cfhi + cghi + defg + defh + defi + degh + degi + dehi + dfgh + dfgi + dfhi + dghi + efgh + efgi + efhi + eghi + fghi
5	5	abcde
6	5	abcde + abcdf + abcef + abdef + acdef + bcdef
7	5	abcde + abcdf + abcdg + abcef + abceg + abcfg + abdef + abdeg + abdfg + abefg + acdef + acdeg + acdfg + acefg + adefg + bcdef + bcdeg + bcdfg + bcefg + bdefg + cdefg
8	5	abcde + abcdf + abcdg + abcdh + abcef + abceg + abceh + abcfg + abcfh + abcgh + abdef + abdeg + abdeh + abdfg + abdfh + abdgh + abefg + abefh + abegh + abfgh + acdef + acdeg + acdeh + acdfg + acdfh + acdgh + acefg + acefh + acegh + acfgh + adefg + adefh + adegh + adfgh + aefgh + bcdef + bcdeg + bcdeh + bcdfg + bcdfh + bcdgh + bcefg + bcefh + bcegh + bcfgh + bdefg + bdefh + bdegh +

Input Variables	Threshold Level	Boolean Function
		bdfgh + befgh + cdefg + cdefh + cdegh + cdfgh + cefgh + defgh
9	5	abcde + abcdf + abcdg + abcdh + abcdi + abcef + abceg + abceh + abcei + abcfg + abcfh + abcfi + abcgh + abcgi + abchi + abdef + abdeg + abdeh + abdei + abdfg + abdfh + abdfi + abdgh + abdgi + abdhi + abefg + abefh + abefi + abegh + abegi + abehi + abfgh + abfgi + abfhi + abghi + acdef + acdeg + acdeh + acdei + acdfg + acdfh + acdfi + acdgh + acdgi + acdhi + acefg + acefh + acefi + acegh + acegi + acehi + acfgh + acfgi + acfhi + acghi + adefg + adefh + adefi + adegh + adegi + adehi + adfgh + adfgi + adfhi + adghi + aefgh + aefgi + aefhi + aeghi + afghi + bcdef + bcdeg + bcdeh + bcdei + bcdfg + bcdfh + bcdfi + bcdgh + bcdgi + bcdhi + bcefg + bcefh + bcefi + bcegh + bcegi + bcehi + bcfgh + bcfgi + bcfhi + bcghi + bdefg + bdefh + bdefi + bdegh + bdegi + bdehi + bdfgh + bdfgi + bdfhi + bdghi + befgh + befgi + befhi + beghi + bfghi + cdefg + cdefh + cdefi + cdegh + cdegi + cdehi + cdfgh + cdfgi + cdfhi + cdghi + cefgh + cefgi + cefhi + ceghi + cfghi + defgh + defgi + defhi + deghi + dfghi + efghi
6	6	abcdef
7	6	abcdef + abcdeg + abcdfg + abcefg + abdefg + acdefg + bcdefg
8	6	abcdef + abcdeg + abcdeh + abcdfg + abcdfh + abcdgh + abcefg + abcefh + abcegh + abcfgh + abdefg + abdefh + abdegh + abdfgh + abefgh + acdefg + acdefh + acdegh + acdfgh + acefgh + adefgh + bcdefg + bcdefh + bcdegh + bcdfgh + bcefgh + bdefgh + cdefgh
9	6	abcdef + abcdeg + abcdeh + abcdei + abcdfg + abcdfh + abcdfi + abcdgh + abcdgi + abcdhi + abcefg + abcefh + abcefi + abcegh + abcegi + abcehi + abcfgh + abcfgi + abcfhi + abcghi + abdefg + abdefh + abdefi + abdegh + abdegi + abdehi + abdfgh + abdfgi + abdfhi + abdghi + abefgh + abefgi + abefhi + abeghi + abfghi + acdefg + acdefh + acdefi + acdegh + acdegi + acdehi + acdfgh + acdfgi + acdfhi + acdghi + acefgh + acefgi + acefhi + aceghi + acfghi + adefgh + adefgi + adefhi + adeghi + adfghi + aefghi + bcdefg + bcdefh + bcdefi + bcdegh + bcdegi + bcdehi + bcdfgh + bcdfgi + bcdfhi + bcdghi + bcefgh + bcefgi + bcefhi + bceghi + bcfghi + bdefgh + bdefgi + bdefhi + bdeghi + bdfghi + befghi + cdefgh + cdefgi + cdefhi + cdeghi + cdfghi + cefghi + defghi
7	7	abcdefg
8	7	abcdefg + abcdefh + abcdegh + abcdfgh + abcefgh + abdefgh + acdefgh + bcdefgh
9	7	abcdefg + abcdefh + abcdefi + abcdegh + abcdegi + abcdehi + abcdfgh + abcdfgi + abcdfhi + abcdghi + abcefgh + abcefgi + abcefhi + abceghi + abcfghi + abdefgh + abdefgi + abdefhi + abdeghi + abdfghi + abefghi + acdefgh + acdefgi + acdefhi + acdeghi + acdfghi + acefghi + adefghi + bcdefgh + bcdefgi + bcdefhi + bcdeghi + bcdfghi + bcefghi + bdefghi + cdefghi
8	8	abcdefgh
9	8	abcdefgh + abcdefgi + abcdefhi + abcdeghi + abcdfghi + abcefghi + abdefghi + acdefghi + bcdefghi
9	9	abcdefghi

A.3 Database of Optimal Networks

The optimal networks provided here use NAND gates with a fan-in limit of 2 as building blocks. The optimality of the networks is proven based on the number of gates in the network. Only one optimal network is given per function even though some functions may have multiple networks of optimal cost.

2 Input Representative Functions from P-Equivalence Class

Functions		Minimum Cost	Number of Optimal Networks	Optimal Network
Binary	Integer			
0001	1	2	1	O = N(I1) I1 = N(a, b)
0010	2	3	1	O = N(I1) I1 = N(I2, a) I2 = N(b)
0110	6	4	1	O = N(I1, I3) I1 = N(I2, b) I2 = N(a, b) I3 = N(a, I2)
0111	7	3	1	O = N(I1, I2) I1 = N(b) I2 = N(a)
1000	8	4	1	O = N(I1) I1 = N(I2, I3) I2 = N(b) I3 = N(a)
1001	9	5	3	O = N(I1) I1 = N(I2, I4) I2 = N(b, I3) I3 = N(b, a) I4 = N(a, I3)
1011	11	2	1	O = N(I1, b) I1 = N(a)
1110	14	1	1	O = N(a, b)

3 Input Representative Functions from P-Equivalence Class

Functions		Minimum Cost	Number of Optimal Networks	Optimal Network
Binary	Integer			
00000001	1	4	1	O = N(I1) I1 = N(a, I2) I2 = N(I3) I3 = N(b, c)
00000010	2	5	2	O = N(I1) I1 = N(I2, I3) I2 = N(c) I3 = N(I4) I4 = N(a, b)
00000110	6	6	4	O = N(I1) I1 = N(I2, I3) I2 = N(b, c) I3 = N(I4, I5) I4 = N(a, c) I5 = N(b, a)
00000111	7	3	1	O = N(I1, I2) I1 = N(a, c) I2 = N(a, b)
00001000	8	6	2	O = N(I1) I1 = N(I2, I3) I2 = N(c) I3 = N(I4) I4 = N(I5, a) I5 = N(b)
00001001	9	7	13	O = N(I1) I1 = N(I2, I4) I2 = N(c, I3) I3 = N(c, b) I4 = N(I5) I5 = N(I6, a) I6 = N(b, I3)
00001011	11	4	2	O = N(I1) I1 = N(I2, a) I2 = N(c, I3) I3 = N(b)
00001110	14	3	1	O = N(I1) I1 = N(I2, a) I2 = N(b, c)
00010110	22	8	2	O = N(I1, I6) I1 = N(I2, I3) I2 = N(a, b) I3 = N(I4, I5) I4 = N(b, c) I5 = N(a, c) I6 = N(I7, I4) I7 = N(I2)
00010111	23	6	4	O = N(I1, I5) I1 = N(b, I2) I2 = N(I3, I4) I3 = N(c) I4 = N(a) I5 = N(a, c)
00011000	24	7	2	O = N(I1) I1 = N(I2, I3) I2 = N(a, b) I3 = N(I4, I5) I4 = N(b, c) I5 = N(a, I6) I6 = N(c)

Functions		Minimum Cost	Number of Optimal Networks	Optimal Network
Binary	Integer			
00011001	25	7	11	O = N(I1, I2) I1 = N(b, c) I2 = N(I3, I4) I3 = N(c) I4 = N(I5) I5 = N(I6, a) I6 = N(b)
00011010	26	6	5	O = N(I1) I1 = N(I2, I3) I2 = N(a, c) I3 = N(I4, I5) I4 = N(b, c) I5 = N(a)
00011011	27	4	1	O = N(I1, I2) I1 = N(b, c) I2 = N(I3, a) I3 = N(c)
00011110	30	6	5	O = N(I1, I5) I1 = N(I2, I3) I2 = N(a, b) I3 = N(I4, I5) I4 = N(b, c) I5 = N(a, I4)
00011111	31	3	1	O = N(I1, I2) I1 = N(b, c) I2 = N(a)
00101000	40	7	4	O = N(I1) I1 = N(I2, I3) I2 = N(c) I3 = N(I4, I6) I4 = N(I5, b) I5 = N(a, b) I6 = N(a, I5)
00101001	41	8	5	O = N(I1, I9) I1 = N(I2, I4) I2 = N(c, I3) I3 = N(a, b) I4 = N(I5, I7) I5 = N(I3, b) I7 = N(a, I3)
00101010	42	4	1	O = N(I1, I3) I1 = N(I2, b) I2 = N(c) I3 = N(a, I2)
00101011	43	5	2	O = N(I1, I4) I1 = N(I2, b) I2 = N(c, I3) I3 = N(c, a) I4 = N(a, I3)
00101100	44	7	13	O = N(I1) I1 = N(I2, I3) I2 = N(c, b) I3 = N(I4, I6) I4 = N(I5, b) I5 = N(a, b) I6 = N(a, I5)
00101101	45	7	5	O = N(I1, I5) I1 = N(I2, I8) I2 = N(I3, I5) I3 = N(I4, b) I4 = N(c) I5 = N(a, I3)

Functions		Minimum Cost	Number of Optimal Networks	Optimal Network
Binary	Integer			
00101110	46	4	1	O = N(I1, I3) I1 = N(I2, b) I2 = N(c, b) I3 = N(a, I2)
00101111	47	4	1	O = N(I1, I3) I1 = N(I2, b) I2 = N(c) I3 = N(a)
00111101	61	6	2	O = N(I1, I4) I1 = N(I2, b) I2 = N(a, I3) I3 = N(c) I4 = N(a, I5) I5 = N(b)
00111110	62	5	2	O = N(I1, I3) I1 = N(I2, b) I2 = N(a, c) I3 = N(a, I4) I4 = N(b)
01101000	104	10	47	O = N(I1, I13) I1 = N(I2, I8) I2 = N(b, I3) I3 = N(I4, I6) I4 = N(a, I5) I5 = N(a, c) I6 = N(I5, c) I8 = N(I6, I11)
01101001	105	8	1	O = N(I1, I13) I1 = N(I2, I3) I2 = N(b, I3) I3 = N(I4, I6) I4 = N(I5, c) I5 = N(a, c) I6 = N(a, I5)
01101010	106	7	1	O = N(I1, I6) I1 = N(I2, c) I2 = N(I3, c) I3 = N(I4, I5) I4 = N(b) I5 = N(a) I6 = N(I3, I2)
01101011	107	8	2	O = N(I1, I12) I1 = N(I2, I6) I2 = N(b, I3) I3 = N(b, I4) I4 = N(I5, c) I5 = N(a, c) I6 = N(I4, I3)
01101110	110	7	9	O = N(I1, I3) I1 = N(I2, c) I2 = N(b, c) I3 = N(I4) I4 = N(I5, I7) I5 = N(a, I2)
01101111	111	7	7	O = N(I1, I3) I1 = N(I2, c) I2 = N(b, c) I3 = N(I4) I4 = N(I5, I6) I5 = N(a) I6 = N(b, I2)

Functions		Minimum Cost	Number of Optimal Networks	Optimal Network
Binary	Integer			
01111110	126	7	1	O = N(I1, I3) I1 = N(I2, c) I2 = N(a) I3 = N(I4, I7) I4 = N(I5, I2) I5 = N(b)
01111111	127	6	1	O = N(I1, I2) I1 = N(c) I2 = N(I3) I3 = N(I4, I5) I4 = N(b) I5 = N(a)
10000000	128	7	1	O = N(I1) I1 = N(I2, I3) I2 = N(c) I3 = N(I4) I4 = N(I5, I6) I5 = N(b) I6 = N(a)
10000001	129	8	2	O = N(I1) I1 = N(I2, I4) I2 = N(c, I3) I3 = N(b) I4 = N(I5, I8) I5 = N(I3, I7) I7 = N(a)
10000010	130	8	13	O = N(I1) I1 = N(I2, I3) I2 = N(c) I3 = N(I4) I4 = N(I5, I7) I5 = N(b, I6) I6 = N(b, a) I7 = N(a, I6)
10000011	131	8	24	O = N(I1) I1 = N(I2, I4) I2 = N(c, I3) I3 = N(b, a) I4 = N(I5) I5 = N(I6, I8) I6 = N(b, I3)
10000110	134	9	18	O = N(I1) I1 = N(I2, I4) I2 = N(c, I3) I3 = N(c, a) I4 = N(I5, I11) I5 = N(I6, I7) I6 = N(b, I7) I7 = N(a, I3)
10000111	135	8	5	O = N(I1, I6) I1 = N(I2, I10) I2 = N(I3, I6) I3 = N(I4, I5) I4 = N(c) I5 = N(b) I6 = N(a, I3)
10001001	137	6	3	O = N(I1) I1 = N(I2, I4) I2 = N(c, I3) I3 = N(b, a) I4 = N(b, I5) I5 = N(c)

Functions		Minimum Cost	Number of Optimal Networks	Optimal Network
Binary	Integer			
10001010	138	5	2	O = N(I1) I1 = N(I2, I3) I2 = N(c) I3 = N(b, I4) I4 = N(a)
10001011	139	5	4	O = N(I1) I1 = N(I2, I4) I2 = N(c, I3) I3 = N(a, b) I4 = N(b, I3)
10001110	142	6	7	O = N(I1) I1 = N(I2, I5) I2 = N(c, I3) I3 = N(a, I4) I4 = N(a, b) I5 = N(b, I4)
10001111	143	5	3	O = N(I1) I1 = N(I2, I4) I2 = N(c, I3) I3 = N(a) I4 = N(b, I3)
10010110	150	9	7	O = N(I1, I11) I1 = N(I2, I5) I2 = N(c, I3) I3 = N(b, I4) I4 = N(b, a) I5 = N(I6, I11) I6 = N(I3, I9)
10010111	151	9	7	O = N(I1, I6) I1 = N(I2, I9) I2 = N(c, I3) I3 = N(c, I4) I4 = N(I5, I7) I5 = N(b, I6) I6 = N(b, a) I7 = N(a, I6)
10011000	152	6	1	O = N(I1, I7) I1 = N(I2, I4) I2 = N(c, I3) I3 = N(a, b) I4 = N(b, I2)
10011010	154	6	1	O = N(I1, I7) I1 = N(I2, I3) I2 = N(c, I3) I3 = N(b, I4) I4 = N(a)
10011011	155	6	3	O = N(I1, I6) I1 = N(I2, I4) I2 = N(c, I3) I3 = N(c, b) I4 = N(b, I3)
10011110	158	8	4	O = N(I1, I7) I1 = N(I2, I5) I2 = N(c, I3) I3 = N(b, I4) I4 = N(a) I5 = N(b, I6) I6 = N(c, b) I7 = N(a, I6)
10011111	159	6	2	O = N(I1, I6) I1 = N(I2, I4) I2 = N(c, I3) I3 = N(c, b) I4 = N(b, I3)

Functions		Minimum Cost	Number of Optimal Networks	Optimal Network
Binary	Integer			
10101000	168	4	1	O = N(I1) I1 = N(I2, I3) I2 = N(c) I3 = N(a, b)
10101001	169	5	1	O = N(I1, I5) I1 = N(I2, I3) I2 = N(c, I3) I3 = N(a, b)
10101011	171	2	1	O = N(I1, c) I1 = N(a, b)
10101100	172	5	2	O = N(I1) I1 = N(I2, I4) I2 = N(c, I3) I3 = N(a) I4 = N(a, b)
10101101	173	5	1	O = N(I1, I4) I1 = N(I2, I3) I2 = N(c) I3 = N(a, b) I4 = N(c, a)
10101110	174	3	1	O = N(I1, c) I1 = N(I2, a) I2 = N(b)
10111100	188	6	3	O = N(I1, I5) I1 = N(I2, I4) I2 = N(c, I3) I3 = N(b) I4 = N(a, b) I5 = N(a, I4)
10111101	189	6	1	O = N(I1, I5) I1 = N(I2, I4) I2 = N(c, I3) I3 = N(b) I4 = N(a, b) I5 = N(a, c)
10111110	190	6	7	O = N(I1, I3) I1 = N(I2, b) I2 = N(a, b) I3 = N(I4) I4 = N(c, I5) I5 = N(a, I2)
10111111	191	5	2	O = N(I1, I2) I1 = N(b) I2 = N(I3) I3 = N(I4, c) I4 = N(a)
11101000	232	7	8	O = N(I1) I1 = N(I2, I3) I2 = N(b, c) I3 = N(a, I4) I4 = N(I5, I6) I5 = N(c) I6 = N(b)
11101001	233	8	9	O = N(I1, I9) I1 = N(I2, I3) I2 = N(b, c) I3 = N(a, I4) I4 = N(I5, I7) I5 = N(c, I2) I7 = N(b, I2)
11101010	234	4	2	O = N(I1) I1 = N(I2, I3) I2 = N(b, c) I3 = N(a, c)

Functions		Minimum Cost	Number of Optimal Networks	Optimal Network
Binary	Integer			
11101011	235	5	2	O = N(I1, c) I1 = N(I2, I4) I2 = N(b, I3) I3 = N(b, a) I4 = N(a, I3)

Functions		Minimum Cost	Number of Optimal Networks	Optimal Network
Binary	Integer			
11101111	239	4	2	O = N(I1, I2) I1 = N(a) I2 = N(I3) I3 = N(b, c)
11111110	254	3	1	O = N(a, I1) I1 = N(I2) I2 = N(b, c)

4 Input Representative Functions from P-Equivalence Class

For these functions we give only the optimal cost and integer representation to preserve space.

Optimal Cost	Number of Functions	Function Index
3	2	855, 43774
4	12	171, 239, 939, 1007, 3822, 3839, 43691, 43759, 61152, 61162, 61167, 65262
5	42	7, 14, 31, 174, 191, 254, 427, 495, 511, 943, 991, 2747, 2766, 2767, 2783, 2798, 2814, 3003, 3055, 3071, 3838, 4094, 34954, 34959, 43178, 43179, 43180, 43183, 43260, 43261, 43263, 43694, 43711, 44031, 60142, 60143, 60159, 61154, 61171, 61178, 61182, 65534
6	108	1, 11, 27, 42, 46, 47, 127, 168, 234, 287, 319, 431, 582, 599, 682, 718, 719, 735, 750, 766, 767, 854, 863, 938, 974, 1022, 1911, 1967, 1983, 2047, 2731, 2746, 2782, 2799, 2811, 3002, 3039, 3066, 3067, 4081, 4091, 8191, 34953, 34955, 34984, 34987, 34988, 34991, 35002, 35007, 35064, 35066, 35067, 35498, 35514, 35515, 35530, 35534, 35535, 35551, 35578, 35582, 35583, 35771, 35807, 35839, 36863, 39322, 39323, 39327, 43181, 43199, 43244, 43246, 43247, 43262, 43688, 43692, 43710, 43720, 43726, 43727, 43754, 43756, 43772, 43773, 43980, 43983, 43997, 44014, 44015, 44028, 44030, 44284, 44285, 44286, 44287, 44541, 44782, 44798, 44799, 45052, 45054, 60139, 60158, 61155, 61179, 65279
7	244	2, 23, 26, 43, 61, 62, 111, 137, 138, 139, 143, 169, 172, 173, 235, 303, 393, 425, 426, 447, 491, 559, 575, 583, 590, 591, 598, 607, 650, 651, 683, 686, 687, 714, 746, 751, 859, 904, 906, 907, 911, 936, 937, 942, 959, 968, 971, 973, 989, 1003, 1005, 1006, 1021, 1638, 1655, 1727, 1774, 1790, 1791, 1935, 1962, 1963, 2031, 2032, 2035, 2039, 2040, 2042, 2043, 2046, 2184, 2216, 2218, 2219, 2235, 2296, 2298, 2299, 2303, 2734, 2751, 2760, 2762, 2794, 2808, 2812, 2813, 3054, 3070, 3808, 3818, 3823, 3824, 3825, 3834, 3835, 4082, 4087, 4088, 6906, 6907, 6910, 6911, 7099, 7163, 7167, 7920, 7934, 7935, 8176, 8177, 8186, 8187, 8190, 10922, 10986, 10990, 10991, 11007, 12014, 12015, 12030, 12031, 12287, 16126, 16127, 34944, 34968, 34971, 34975, 34989, 34990, 35000, 35001, 35004, 35055, 35065, 35070, 35241, 35243, 35259, 35311, 35323, 35327, 35499, 35503, 35519, 35528, 35531, 35546, 35549, 35550, 35566, 35567, 35576, 35579, 35580, 35823, 35835, 36590, 36606, 36607, 36856, 39320, 39337, 39339, 39353, 39357, 39359, 39417, 39419, 39899, 39903, 43144, 43177, 43182, 43208, 43212, 43245, 43437, 43517, 43689, 43693, 43708, 43722, 43723, 43725, 43736, 43737, 43740, 43743, 43755, 43757, 43979, 43996, 43999, 44010, 44011, 44012, 44029, 44268, 44269, 44270, 44271, 44280, 44525, 44540, 44543, 44778, 44780, 44783, 44784, 44785, 44787, 44789, 44794, 44795, 44796, 44797, 45041, 45043, 45048, 45051, 45053, 60074, 60096, 60104, 60106, 60136, 60140, 60394, 60399, 61153, 61158, 61160, 61163, 61170, 61176, 61434, 61438, 61439, 65256, 65258, 65263
8	487	6, 8, 9, 25, 30, 44, 110, 142, 152, 154, 155, 159, 188, 189, 190, 279, 283, 299, 317, 392, 394, 395, 399, 409, 411, 424, 429, 430, 494, 510, 554, 558, 597, 602, 606, 639, 648, 654, 655, 667, 671, 703, 706, 710, 711, 712, 716, 727, 730, 734, 858, 862, 895, 905, 908, 910, 921, 923, 927, 940, 941, 962, 964, 965, 966, 967, 969, 970, 981, 983, 985, 986, 987, 988, 990, 1002, 1004, 1654, 1695, 1702, 1710, 1711, 1782, 1783, 1910, 1931, 1951, 1955, 1957, 1959, 1966, 1978, 1979, 2019, 2023, 2030, 2034, 2186, 2187, 2191, 2220, 2222, 2223, 2232, 2234, 2239, 2282, 2286, 2287, 2302, 2457, 2473, 2475, 2479, 2489, 2491, 2543, 2553, 2555, 2559, 2722, 2723, 2728, 2732, 2736, 2739, 2744, 2750, 2752, 2754, 2758, 2759, 2763, 2768, 2775, 2776, 2778, 2779, 2780, 2781, 2784, 2788, 2795, 2796, 2801, 2802, 2803, 2804, 2806, 2807, 2992, 2993, 2995, 3007, 3033, 3038, 3040, 3045, 3050, 3051, 3056, 3057, 3059, 3060, 3061, 3063, 3064, 3068, 3069, 3810, 3826, 3827, 4086, 6330, 6331, 6394, 6395, 6398, 6399, 6553, 6570, 6585, 6587, 6591, 6649, 6651, 6655, 6842, 6843, 6846, 6847, 6874, 6876, 6878, 6879, 6896, 7089, 7098, 7103, 7129, 7131, 7135, 7140, 7150, 7151, 7152, 7153, 7156, 7157, 7162, 7164, 7165, 7166, 7922, 7930, 8178, 8179, 10410, 10411, 10475, 10479, 10492, 10493, 10495, 10926, 10927, 10943, 10956, 10958, 10959, 10965, 10973, 10975, 11004, 11005, 11006, 11246, 11247, 11263, 11500, 11502, 11503, 11516, 11517, 11518, 11519, 11745, 11757, 11759, 11760, 11772, 11773, 11775, 12002, 12016, 12017, 12018, 12019, 12027, 12029, 12272, 12273, 12274, 12275, 12277, 12279, 12282, 12283, 12284, 12285, 12286, 15580, 15599, 15613, 15614, 15837, 15868, 15869, 15871, 16110, 16381, 16382, 32898, 32899, 32904, 32906, 32907, 32910, 32911, 32938, 32942, 32943, 32959, 33023, 34945, 34946, 34947, 34951, 34958, 34960, 34961, 34970, 34976, 34978, 34979, 34985, 34992, 34994, 34995, 35006, 35050, 35056, 35059, 35063, 35225, 35227, 35242, 35247, 35257, 35258, 35263, 35320, 35321, 35322, 35502, 35518, 35544, 35545, 35547, 35548, 35562, 35581, 35770, 35775, 35802, 35803, 35822, 35832, 35834, 35836, 35838, 36858, 36859, 36862, 39048, 39065, 39066, 39067, 39071, 39096, 39098, 39099, 39103, 39160, 39162, 39166, 39167, 39312, 39313, 39315, 39336, 39341, 39343, 39352, 39354, 39356, 39416, 39422, 39642, 39645, 39646, 39647, 39867, 39897, 39931, 39935, 43145, 43146, 43147, 43148, 43151, 43160, 43161, 43163, 43196, 43197, 43198, 43210, 43215, 43224, 43228, 43229, 43231, 43240, 43417, 43432, 43434, 43435, 43436, 43439, 43453, 43485, 43501, 43503, 43516, 43519, 43709, 43712, 43716, 43721, 43733, 43738, 43739, 43742, 43752, 43964, 43967, 43976, 43977, 43981, 43982, 44013, 44236, 44252, 44253, 44256, 44272, 44282, 44283, 44509, 44517, 44524, 44527, 44528, 44533, 44536, 44542, 44768, 44770, 44786, 44791, 45042, 45044, 45047, 48380, 48381, 48383, 48894, 48895, 49151, 59626, 59627, 59630, 59631, 59646, 59647, 60075, 60078,

Optimal Cost	Number of Functions	Function Index
		60079, 60095, 60098, 60105, 60107, 60110, 60111, 60117, 60120, 60121, 60122, 60124, 60125, 60127, 60141, 60156, 60157, 60377, 60392, 60396, 60398, 60415, 61159, 61174, 61175, 61177, 61431, 61432, 61435
9	728	24, 40, 45, 106, 126, 128, 130, 131, 232, 281, 282, 286, 298, 301, 302, 316, 318, 367, 383, 415, 428, 445, 489, 490, 555, 574, 576, 578, 579, 586, 596, 603, 605, 622, 623, 642, 649, 652, 666, 680, 684, 702, 704, 707, 708, 715, 717, 726, 731, 732, 733, 747, 748, 764, 765, 830, 856, 857, 878, 879, 892, 894, 898, 899, 909, 920, 922, 925, 958, 961, 984, 1000, 1639, 1678, 1679, 1706, 1719, 1726, 1762, 1766, 1775, 1778, 1779, 1786, 1919, 1928, 1930, 1934, 1952, 1954, 1956, 1958, 1960, 1964, 1965, 1968, 1970, 1971, 1973, 1975, 1980, 1982, 2022, 2026, 2027, 2033, 2038, 2041, 2185, 2200, 2201, 2203, 2208, 2217, 2221, 2224, 2233, 2236, 2238, 2280, 2283, 2288, 2297, 2458, 2459, 2463, 2474, 2478, 2490, 2492, 2493, 2495, 2539, 2552, 2554, 2721, 2729, 2733, 2737, 2738, 2745, 2755, 2761, 2769, 2770, 2771, 2773, 2774, 2777, 2785, 2786, 2787, 2789, 2790, 2792, 2797, 2809, 2994, 3000, 3001, 3004, 3006, 3024, 3027, 3030, 3031, 3034, 3035, 3041, 3043, 3044, 3052, 3053, 3058, 3062, 3065, 3809, 3811, 3819, 3830, 3831, 3832, 4089, 5886, 6043, 6047, 6058, 6063, 6079, 6126, 6127, 6143, 6296, 6298, 6328, 6334, 6335, 6392, 6393, 6554, 6555, 6559, 6569, 6571, 6574, 6575, 6586, 6588, 6589, 6638, 6648, 6650, 6654, 6826, 6832, 6833, 6858, 6862, 6863, 6875, 6877, 6890, 6894, 6897, 6901, 6904, 6905, 6908, 6909, 7088, 7091, 7097, 7101, 7120, 7121, 7130, 7134, 7136, 7141, 7146, 7147, 7148, 7149, 7154, 7155, 7158, 7159, 7160, 7161, 7918, 7921, 7923, 7931, 8183, 8184, 8185, 10408, 10414, 10415, 10430, 10476, 10478, 10494, 10923, 10942, 10944, 10948, 10954, 10955, 10957, 10972, 10987, 10988, 11210, 11212, 11214, 11215, 11220, 11228, 11229, 11230, 11231, 11242, 11243, 11244, 11260, 11261, 11262, 11501, 11504, 11508, 11756, 11761, 11765, 11768, 12000, 12003, 12010, 12012, 12021, 12023, 12026, 12028, 12276, 12278, 12280, 15561, 15565, 15581, 15583, 15595, 15596, 15597, 15598, 15852, 15853, 15855, 16111, 16124, 16125, 28398, 28414, 28415, 28671, 32767, 32897, 32903, 32905, 32920, 32921, 32922, 32927, 32936, 32939, 32940, 32941, 33002, 33006, 33007, 33022, 33152, 33160, 33161, 33162, 33163, 33167, 33194, 33195, 33199, 33215, 33263, 33279, 33408, 33411, 33414, 33415, 33416, 33418, 33419, 33420, 33422, 33423, 33450, 33454, 33455, 33470, 33471, 33472, 33474, 33475, 33478, 33479, 33484, 33486, 33487, 33495, 33503, 33518, 33534, 33535, 33664, 33665, 33666, 33671, 33672, 33673, 33674, 33675, 33676, 33679, 33706, 33707, 33711, 33727, 33728, 33730, 33731, 33734, 33735, 33740, 33743, 33751, 33759, 33775, 33791, 34688, 34690, 34691, 34696, 34698, 34699, 34703, 34725, 34726, 34727, 34730, 34735, 34740, 34741, 34743, 34751, 34800, 34803, 34807, 34815, 34963, 34977, 34980, 34981, 34983, 34993, 34996, 34997, 34999, 35005, 35040, 35042, 35043, 35048, 35051, 35057, 35058, 35224, 35226, 35231, 35232, 35233, 35235, 35240, 35245, 35246, 35251, 35256, 35260, 35261, 35307, 35310, 35312, 35313, 35326, 35488, 35490, 35491, 35496, 35500, 35504, 35505, 35506, 35507, 35512, 35513, 35516, 35520, 35522, 35523, 35529, 35536, 35539, 35541, 35543, 35552, 35563, 35564, 35568, 35570, 35571, 35573, 35575, 35577, 35760, 35763, 35768, 35772, 35792, 35794, 35795, 35799, 35800, 35801, 35806, 35819, 35820, 35824, 35826, 35827, 35828, 35829, 35831, 35833, 35837, 36591, 36602, 36603, 36848, 36850, 36851, 36855, 36857, 39049, 39051, 39055, 39056, 39057, 39058, 39059, 39070, 39080, 39082, 39083, 39086, 39087, 39097, 39102, 39161, 39163, 39314, 39318, 39319, 39326, 39328, 39329, 39330, 39331, 39333, 39334, 39335, 39340, 39342, 39344, 39345, 39346, 39347, 39348, 39349, 39351, 39358, 39407, 39408, 39409, 39410, 39411, 39414, 39415, 39418, 39594, 39608, 39610, 39611, 39615, 39626, 39630, 39631, 39640, 39641, 39643, 39644, 39674, 39678, 39679, 39864, 39865, 39866, 39868, 39871, 39902, 39918, 39919, 39929, 39932, 40959, 43149, 43150, 43165, 43167, 43211, 43213, 43214, 43225, 43226, 43230, 43242, 43243, 43418, 43419, 43420, 43421, 43423, 43452, 43455, 43465, 43468, 43469, 43471, 43481, 43487, 43497, 43500, 43502, 43518, 43714, 43715, 43717, 43719, 43735, 43965, 43966, 43968, 43971, 43972, 43974, 43975, 43978, 43988, 43989, 43991, 43992, 43993, 43994, 43995, 43998, 44008, 44232, 44234, 44239, 44255, 44259, 44264, 44266, 44273, 44275, 44276, 44277, 44279, 44281, 44506, 44508, 44511, 44512, 44513, 44519, 44526, 44529, 44531, 44532, 44535, 44537, 44538, 44539, 44769, 44771, 44772, 44773, 44774, 44776, 44779, 44781, 44788, 44790, 44792, 45046, 45049, 48332, 48348, 48364, 48366, 48367, 48382, 48605, 48620, 48636, 48637, 48639, 48878, 48879, 48892, 49148, 49149, 49150, 59560, 59562, 59625, 59883, 59887, 60094, 60097, 60099, 60102, 60103, 60118, 60119, 60123, 60126, 60137, 60350, 60351, 60375, 60376, 60379, 60383, 60393, 60397, 60412, 60413, 60414, 61161, 61430, 61433, 65259
10	958	22, 41, 104, 105, 107, 129, 134, 135, 158, 233, 280, 296, 297, 300, 361, 363, 366, 384, 385, 387, 398, 408, 410, 444, 446, 488, 556, 557, 572, 573, 577, 584, 585, 587, 600, 604, 638, 640, 643, 646, 647, 653, 664, 665, 670, 681, 685, 705, 709, 713, 725, 728, 729, 744, 745, 749, 829, 872, 874, 875, 876, 893, 896, 897, 901, 903, 924, 956, 957, 980, 982, 1001, 1632, 1634, 1635, 1642, 1646, 1647, 1650, 1651, 1658, 1662, 1663, 1672, 1674, 1675, 1689, 1691, 1696, 1698, 1699, 1700, 1701, 1703, 1704, 1705, 1707, 1708, 1709, 1712, 1715, 1717, 1718, 1722, 1723, 1725, 1760, 1763, 1767, 1768, 1770, 1776, 1777, 1784, 1785, 1787, 1914, 1915, 1918, 1920, 1923, 1927, 1929, 1945, 1946, 1947, 1953, 1961, 1969, 1972, 1976, 1977, 1981, 2016, 2017, 2018, 2024, 2025, 2176, 2190, 2202, 2207, 2210, 2211, 2212, 2213, 2227, 2237, 2272, 2274, 2289, 2290, 2291, 2295, 2456, 2465, 2467, 2472, 2476, 2477, 2480, 2481, 2488, 2494, 2529, 2537, 2538, 2542, 2544, 2545, 2547, 2558, 2724, 2726, 2727, 2740, 2741, 2743, 2748, 2749, 2753, 2772, 2791, 2996, 2997, 2999, 3005, 3025, 3026, 3032, 3042,

Optimal Cost	Number of Functions	Function Index
		3046, 3047, 3048, 3049, 3814, 3815, 3816, 3833, 5786, 5790, 5802, 5806, 5822, 5823, 5866, 5870, 5887, 6024, 6025, 6026, 6027, 6030, 6031, 6041, 6042, 6046, 6056, 6059, 6060, 6061, 6062, 6076, 6077, 6078, 6120, 6122, 6123, 6142, 6280, 6297, 6299, 6312, 6314, 6315, 6318, 6319, 6323, 6329, 6332, 6378, 6382, 6384, 6387, 6552, 6568, 6572, 6573, 6576, 6577, 6579, 6584, 6590, 6634, 6639, 6640, 6641, 6643, 6816, 6817, 6819, 6821, 6827, 6830, 6831, 6834, 6835, 6837, 6839, 6840, 6841, 6851, 6856, 6859, 6864, 6865, 6866, 6867, 6869, 6872, 6873, 6880, 6884, 6885, 6888, 6891, 6892, 6893, 6895, 6898, 6899, 6900, 6903, 7090, 7092, 7093, 7096, 7100, 7102, 7122, 7123, 7126, 7127, 7128, 7137, 7138, 7139, 7142, 7143, 7144, 7145, 7904, 7905, 7906, 7907, 7914, 7915, 7919, 7926, 7927, 7928, 8182, 10376, 10412, 10413, 10428, 10431, 10440, 10444, 10461, 10472, 10474, 10477, 10665, 10667, 10669, 10671, 10729, 10733, 10735, 10748, 10749, 10751, 10920, 10924, 10925, 10945, 10947, 10949, 10952, 10964, 10967, 10968, 10969, 10970, 10974, 10984, 10989, 11199, 11209, 11211, 11213, 11221, 11223, 11224, 11225, 11226, 11245, 11468, 11469, 11484, 11485, 11488, 11489, 11490, 11491, 11498, 11499, 11505, 11506, 11507, 11509, 11510, 11511, 11512, 11513, 11514, 11515, 11741, 11744, 11747, 11749, 11751, 11752, 11753, 11758, 11762, 11763, 11764, 11767, 11769, 11770, 11771, 11774, 12001, 12006, 12007, 12011, 12013, 12020, 12022, 12024, 12025, 12281, 15554, 15558, 15559, 15560, 15562, 15563, 15566, 15575, 15576, 15577, 15579, 15582, 15592, 15594, 15833, 15836, 15839, 15854, 15870, 16106, 16107, 16108, 16109, 27306, 27328, 27336, 27338, 27370, 27374, 27375, 27391, 27627, 27631, 27647, 28384, 28386, 28387, 28390, 28394, 28399, 28403, 28406, 28410, 28662, 32494, 32510, 32511, 32766, 32768, 32776, 32778, 32779, 32783, 32902, 32923, 32937, 32956, 32957, 32958, 33003, 33154, 33155, 33177, 33179, 33183, 33192, 33193, 33197, 33198, 33258, 33259, 33262, 33409, 33412, 33413, 33417, 33421, 33430, 33431, 33433, 33435, 33439, 33451, 33473, 33476, 33477, 33480, 33481, 33482, 33483, 33485, 33493, 33494, 33497, 33498, 33499, 33500, 33501, 33502, 33514, 33515, 33516, 33519, 33532, 33668, 33669, 33670, 33677, 33678, 33687, 33689, 33691, 33693, 33695, 33704, 33708, 33710, 33724, 33726, 33729, 33732, 33733, 33736, 33737, 33738, 33739, 33741, 33742, 33749, 33750, 33753, 33754, 33755, 33756, 33757, 33758, 33770, 33771, 33772, 33774, 33788, 33789, 33790, 34433, 34434, 34435, 34439, 34446, 34447, 34455, 34468, 34469, 34470, 34471, 34478, 34479, 34485, 34486, 34487, 34494, 34495, 34528, 34543, 34545, 34550, 34551, 34558, 34559, 34689, 34694, 34697, 34707, 34710, 34711, 34720, 34722, 34723, 34724, 34728, 34729, 34731, 34732, 34733, 34736, 34737, 34739, 34742, 34744, 34747, 34748, 34749, 34785, 34786, 34787, 34791, 34795, 34799, 34801, 34802, 34808, 34810, 34811, 34950, 34962, 34966, 34967, 34974, 34982, 34998, 35041, 35046, 35047, 35049, 35062, 35216, 35217, 35219, 35234, 35236, 35237, 35239, 35244, 35248, 35249, 35250, 35253, 35255, 35262, 35296, 35305, 35306, 35314, 35315, 35318, 35319, 35489, 35493, 35495, 35497, 35501, 35509, 35511, 35517, 35521, 35526, 35527, 35537, 35538, 35540, 35542, 35553, 35554, 35555, 35556, 35557, 35560, 35565, 35569, 35572, 35574, 35761, 35762, 35769, 35773, 35774, 35793, 35808, 35809, 35811, 35813, 35815, 35816, 35817, 35818, 35821, 35825, 35830, 36576, 36584, 36586, 36587, 36592, 36593, 36594, 36595, 36599, 36600, 36601, 36849, 38530, 38534, 38535, 38551, 38552, 38553, 38554, 38555, 38558, 38559, 38570, 38590, 38591, 38654, 38655, 38786, 38787, 38790, 38791, 38806, 38809, 38810, 38811, 38815, 38826, 38827, 38831, 38847, 38911, 39040, 39042, 39050, 39072, 39073, 39075, 39076, 39077, 39078, 39081, 39084, 39085, 39088, 39089, 39090, 39091, 39092, 39093, 39095, 39100, 39101, 39146, 39150, 39151, 39152, 39153, 39154, 39155, 39159, 39332, 39350, 39392, 39393, 39395, 39399, 39400, 39401, 39402, 39403, 39589, 39590, 39593, 39595, 39599, 39600, 39601, 39602, 39603, 39605, 39607, 39609, 39612, 39613, 39614, 39616, 39617, 39619, 39624, 39625, 39627, 39632, 39633, 39635, 39636, 39637, 39639, 39662, 39663, 39664, 39665, 39669, 39671, 39672, 39673, 39675, 39676, 39677, 39857, 39859, 39861, 39863, 39869, 39870, 39888, 39889, 39891, 39895, 39896, 39898, 39912, 39914, 39915, 39916, 39917, 39920, 39921, 39923, 39924, 39925, 39926, 39927, 39928, 39930, 39933, 39934, 40696, 40702, 40703, 40944, 40945, 40946, 40947, 40950, 40951, 40952, 40953, 40954, 40955, 40958, 43136, 43139, 43140, 43141, 43162, 43164, 43166, 43200, 43204, 43209, 43221, 43227, 43241, 43416, 43438, 43464, 43467, 43480, 43482, 43483, 43484, 43486, 43496, 43499, 43713, 43718, 43732, 43734, 43753, 43969, 43970, 43973, 44009, 44224, 44235, 44237, 44238, 44240, 44248, 44250, 44254, 44257, 44258, 44260, 44261, 44267, 44274, 44278, 44504, 44505, 44507, 44510, 44515, 44516, 44520, 44521, 44522, 44523, 44530, 44534, 44775, 44793, 48328, 48330, 48331, 48333, 48335, 48349, 48351, 48360, 48361, 48362, 48363, 48365, 48601, 48602, 48603, 48607, 48616, 48621, 48622, 48623, 48638, 48874, 48875, 48876, 48877, 48893, 59528, 59561, 59563, 59564, 59565, 59566, 59567, 59583, 59817, 59880, 59882, 59886, 59903, 60072, 60076, 60077, 60116, 60378, 65257
11	759	150, 151, 278, 362, 382, 386, 390, 391, 414, 552, 601, 618, 619, 620, 636, 637, 641, 644, 645, 660, 662, 663, 668, 669, 700, 701, 724, 873, 877, 900, 902, 916, 917, 918, 919, 926, 1659, 1664, 1666, 1670, 1671, 1673, 1680, 1681, 1682, 1683, 1686, 1687, 1688, 1690, 1694, 1697, 1714, 1716, 1720, 1721, 1724, 1771, 1912, 1921, 1922, 1926, 1936, 1937, 1938, 1939, 1942, 1943, 1944, 1950, 1974, 2177, 2178, 2179, 2183, 2192, 2193, 2194, 2195, 2206, 2209, 2214, 2215, 2225, 2226, 2228, 2229, 2230, 2231, 2273, 2275, 2278, 2279, 2281, 2294, 2448, 2449, 2451, 2464, 2466, 2468, 2469, 2470, 2471, 2482, 2483, 2485, 2528, 2530, 2531, 2535, 2536, 2546, 2550, 2551, 2742, 2793, 5770, 5782, 5785, 5791, 5804, 5807, 5820, 5821, 5871, 6015, 6016, 6017, 6018, 6019, 6022, 6023, 6038, 6039,

Optimal Cost	Number of Functions	Function Index
		6040, 6057, 6272, 6273, 6282, 6283, 6287, 6288, 6289, 6302, 6303, 6304, 6305, 6306, 6307, 6313, 6316, 6317, 6320, 6321, 6322, 6325, 6333, 6368, 6370, 6371, 6375, 6376, 6379, 6383, 6385, 6386, 6390, 6391, 6544, 6545, 6547, 6560, 6561, 6562, 6563, 6564, 6565, 6566, 6567, 6578, 6581, 6583, 6624, 6626, 6627, 6630, 6631, 6632, 6633, 6635, 6642, 6646, 6647, 6818, 6820, 6824, 6825, 6829, 6836, 6838, 6844, 6845, 6848, 6849, 6850, 6855, 6868, 6870, 6871, 6881, 6882, 6883, 6886, 6887, 6889, 6902, 7095, 7910, 7911, 7929, 10368, 10370, 10371, 10373, 10378, 10379, 10380, 10381, 10382, 10383, 10392, 10393, 10396, 10397, 10409, 10429, 10432, 10433, 10434, 10435, 10436, 10437, 10438, 10439, 10441, 10442, 10443, 10445, 10446, 10447, 10452, 10453, 10455, 10456, 10457, 10458, 10459, 10460, 10463, 10473, 10649, 10664, 10666, 10668, 10684, 10685, 10687, 10688, 10689, 10696, 10697, 10700, 10701, 10709, 10712, 10713, 10716, 10717, 10731, 10732, 10734, 10940, 10941, 10946, 10950, 10951, 10953, 10966, 10971, 11196, 11197, 11198, 11200, 11201, 11202, 11203, 11204, 11205, 11207, 11208, 11227, 11240, 11241, 11456, 11457, 11458, 11459, 11464, 11465, 11466, 11470, 11471, 11472, 11473, 11474, 11475, 11476, 11477, 11479, 11480, 11481, 11486, 11487, 11492, 11493, 11494, 11495, 11496, 11497, 11728, 11729, 11730, 11731, 11733, 11735, 11736, 11738, 11740, 11742, 11743, 11746, 11748, 11750, 11754, 11755, 11766, 12004, 12005, 12008, 15553, 15556, 15557, 15572, 15573, 15574, 15578, 15593, 15829, 15831, 15832, 15835, 15838, 15848, 15849, 15850, 15851, 16104, 26760, 26777, 26797, 26856, 26858, 26859, 26862, 26863, 26878, 26879, 27033, 27049, 27050, 27053, 27113, 27115, 27118, 27119, 27135, 27305, 27310, 27311, 27327, 27330, 27331, 27334, 27339, 27342, 27343, 27349, 27352, 27353, 27354, 27355, 27356, 27357, 27358, 27359, 27368, 27371, 27372, 27373, 27388, 27389, 27390, 27609, 27611, 27614, 27626, 27629, 27630, 27646, 28391, 28392, 28395, 28402, 28407, 28409, 28411, 28663, 28664, 28665, 28666, 28667, 28670, 32495, 32769, 32770, 32771, 32775, 32782, 32795, 32799, 32808, 32810, 32812, 32814, 32815, 32828, 32831, 32895, 32918, 32919, 32926, 33000, 33071, 33087, 33158, 33159, 33166, 33175, 33176, 33178, 33196, 33212, 33213, 33214, 33278, 33345, 33347, 33350, 33351, 33358, 33365, 33366, 33367, 33375, 33429, 33432, 33434, 33436, 33437, 33438, 33448, 33452, 33468, 33492, 33496, 33517, 33533, 33623, 33631, 33685, 33686, 33688, 33690, 33692, 33694, 33705, 33709, 33725, 33748, 33752, 33773, 34432, 34440, 34441, 34442, 34443, 34449, 34450, 34451, 34454, 34458, 34462, 34463, 34464, 34465, 34466, 34467, 34474, 34475, 34480, 34482, 34483, 34484, 34530, 34534, 34542, 34544, 34546, 34547, 34679, 34702, 34704, 34705, 34706, 34712, 34713, 34714, 34715, 34718, 34719, 34721, 34734, 34738, 34745, 34746, 34750, 34784, 34790, 34794, 34798, 34806, 34814, 35218, 35223, 35238, 35252, 35297, 35299, 35302, 35304, 35492, 35494, 35508, 35510, 35558, 35559, 35561, 35764, 35765, 35767, 35798, 35810, 35812, 35814, 36577, 36578, 36579, 36585, 36598, 36854, 38536, 38538, 38542, 38543, 38574, 38575, 38792, 38794, 38795, 38799, 38808, 38814, 38829, 38830, 38844, 38845, 38846, 38895, 38910, 39041, 39043, 39054, 39062, 39063, 39074, 39079, 39094, 39138, 39144, 39147, 39158, 39394, 39398, 39584, 39588, 39591, 39592, 39597, 39598, 39604, 39606, 39618, 39623, 39634, 39638, 39648, 39653, 39656, 39658, 39660, 39661, 39666, 39667, 39668, 39670, 39856, 39858, 39860, 39890, 39894, 39904, 39906, 39907, 39908, 39909, 39910, 39911, 39913, 39922, 40686, 40687, 40688, 40691, 40694, 40695, 40697, 40698, 40699, 43137, 43138, 43142, 43143, 43156, 43157, 43159, 43201, 43202, 43203, 43205, 43207, 43220, 43223, 43413, 43454, 43456, 43457, 43459, 43460, 43461, 43463, 43466, 43470, 43477, 43498, 44225, 44227, 44228, 44229, 44231, 44233, 44241, 44243, 44244, 44245, 44247, 44249, 44251, 44262, 44263, 44265, 44496, 44497, 44500, 44501, 44503, 44514, 44518, 44777, 48320, 48323, 48324, 48327, 48329, 48334, 48341, 48344, 48345, 48346, 48347, 48350, 48597, 48600, 48604, 48606, 48617, 48618, 48619, 48872, 59529, 59530, 59531, 59534, 59535, 59544, 59545, 59547, 59551, 59580, 59581, 59582, 59801, 59816, 59818, 59819, 59820, 59821, 59823, 59836, 59837, 59839, 59902, 60073, 60092, 60093, 60348, 60349, 60374, 60382
12	411	360, 406, 407, 553, 616, 617, 621, 661, 1633, 1641, 1643, 1665, 1667, 1713, 1761, 1769, 1913, 2182, 2198, 2199, 2450, 2454, 2455, 2462, 2484, 2486, 2487, 2534, 2998, 3817, 5742, 5743, 5758, 5759, 5762, 5766, 5767, 5768, 5771, 5774, 5775, 5783, 5784, 5787, 5800, 5801, 5803, 5805, 5864, 5865, 5867, 6014, 6121, 6274, 6275, 6281, 6286, 6290, 6291, 6308, 6309, 6311, 6324, 6326, 6327, 6369, 6374, 6377, 6546, 6550, 6551, 6558, 6580, 6582, 6625, 6822, 6823, 6828, 6854, 6857, 7094, 7912, 10369, 10372, 10374, 10375, 10377, 10390, 10394, 10395, 10398, 10399, 10454, 10462, 10644, 10648, 10651, 10652, 10653, 10655, 10670, 10686, 10690, 10691, 10692, 10693, 10694, 10695, 10698, 10699, 10702, 10703, 10708, 10710, 10711, 10715, 10718, 10719, 10728, 10730, 10750, 10921, 10985, 11206, 11222, 11460, 11461, 11462, 11463, 11467, 11478, 11482, 11483, 11732, 11734, 11737, 11739, 12009, 15828, 15830, 15834, 16105, 26776, 26792, 26793, 26794, 26796, 26798, 26799, 26812, 26814, 26857, 27030, 27034, 27039, 27048, 27051, 27054, 27055, 27068, 27069, 27070, 27071, 27112, 27114, 27134, 27304, 27307, 27309, 27326, 27329, 27335, 27337, 27348, 27350, 27351, 27369, 27582, 27583, 27606, 27608, 27610, 27615, 27624, 27625, 27628, 27644, 27645, 28385, 28408, 32488, 32490, 32491, 32774, 32777, 32791, 32792, 32793, 32794, 32798, 32809, 32811, 32813, 32829, 32830, 32874, 32875, 32878, 32879, 32894, 33001, 33047, 33048, 33049, 33050, 33051, 33055, 33064, 33065, 33066, 33067, 33068, 33069, 33070, 33084, 33085, 33086, 33135, 33150, 33151, 33182, 33256, 33257, 33320, 33322, 33323, 33324, 33325, 33326, 33327, 33340, 33341, 33342, 33343, 33344, 33346, 33352, 33353, 33354, 33355, 33359, 33364, 33368, 33370, 33371, 33372, 33373, 33374, 33388, 33390, 33404, 33405, 33406, 33407,

Optimal Cost	Number of Functions	Function Index
		33428, 33449, 33453, 33469, 33512, 33596, 33597, 33599, 33624, 33626, 33627, 33630, 33644, 33646, 33647, 33660, 33661, 33662, 33663, 33684, 33768, 33769, 34406, 34418, 34419, 34422, 34423, 34448, 34457, 34459, 34472, 34473, 34476, 34477, 34481, 34490, 34491, 34492, 34493, 34529, 34531, 34535, 34538, 34539, 34554, 34555, 34680, 34682, 34683, 34687, 34809, 35222, 35230, 35254, 35298, 35303, 35766, 36582, 36583, 38528, 38531, 38537, 38539, 38569, 38571, 38573, 38588, 38589, 38635, 38638, 38639, 38784, 38785, 38793, 38798, 38824, 38825, 38828, 38891, 38894, 39046, 39047, 39136, 39137, 39139, 39142, 39143, 39145, 39585, 39586, 39587, 39596, 39622, 39650, 39651, 39652, 39654, 39655, 39657, 39659, 39862, 39905, 40679, 40681, 40682, 40683, 40689, 40690, 43158, 43206, 43222, 43412, 43414, 43415, 43422, 43458, 43462, 43476, 43478, 43479, 43990, 44226, 44242, 44246, 44498, 44499, 44502, 48321, 48322, 48325, 48326, 48340, 48342, 48343, 48596, 48598, 48599, 48873, 59520, 59546, 59550, 59802, 59803, 59806, 59807, 59822, 59838
13	135	1640, 1656, 1657, 5736, 5738, 5739, 5760, 5761, 5763, 5769, 6278, 6279, 6294, 6295, 6310, 7913, 10388, 10389, 10391, 10645, 10647, 10650, 10654, 10714, 26758, 26761, 26762, 26763, 26766, 26774, 26775, 26778, 26779, 26782, 26783, 26795, 26813, 26815, 27031, 27032, 27035, 27038, 27052, 27308, 27324, 27325, 27581, 27607, 28393, 32489, 32790, 32872, 32873, 33054, 33128, 33130, 33131, 33134, 33174, 33321, 33369, 33384, 33385, 33386, 33387, 33389, 33391, 33513, 33598, 33622, 33625, 33640, 33641, 33642, 33643, 33645, 34400, 34401, 34402, 34403, 34407, 34408, 34409, 34410, 34411, 34414, 34415, 34424, 34425, 34426, 34427, 34430, 34431, 34456, 34488, 34489, 34536, 34537, 34552, 34553, 34678, 34681, 34686, 34792, 34793, 38505, 38506, 38507, 38510, 38511, 38526, 38527, 38529, 38568, 38572, 38633, 38634, 38783, 38888, 38889, 38890, 39649, 40672, 40673, 40674, 40675, 40678, 40680, 44230, 59521, 59522, 59523, 59526, 59527, 59800
14	18	5737, 10646, 26752, 26753, 26754, 26755, 26759, 26767, 27580, 33046, 33129, 38504, 38632, 38782, 59542, 59543, 59798, 59799

5 Input Representative Functions from P-Equivalence Class

For these functions we give only the optimal cost and integer representation to preserve space.

Optimal Cost	Number of Functions	Function Index
5	4	855, 43774, 44031, 65534
6	17	171, 239, 939, 1007, 1911, 2047, 3822, 3839, 8191, 43691, 43759, 44799, 61152, 61162, 61167, 65262, 65279
7	50	7, 14, 31, 174, 191, 254, 427, 431, 495, 511, 863, 943, 991, 2746, 2747, 2766, 2767, 2783, 2798, 2814, 3003, 3055, 3067, 3071, 3838, 4094, 7167, 11007, 12031, 12287, 34954, 34959, 43178, 43179, 43180, 43183, 43260, 43261, 43263, 43694, 43711, 44015, 60142, 60143, 60159, 61154, 61171, 61178, 61182, 61439
8	139	1, 11, 27, 42, 46, 47, 127, 168, 234, 287, 303, 319, 425, 582, 598, 599, 607, 682, 686, 718, 719, 735, 750, 766, 767, 854, 895, 938, 959, 974, 1006, 1022, 1967, 1983, 2035, 2039, 2731, 2782, 2794, 2799, 2811, 3002, 3007, 3039, 3066, 3823, 3824, 4081, 4091, 6911, 7099, 7135, 11263, 16127, 34953, 34955, 34984, 34987, 34988, 34991, 35002, 35007, 35064, 35066, 35067, 35327, 35498, 35514, 35515, 35530, 35534, 35535, 35551, 35578, 35582, 35583, 35771, 35807, 35839, 36863, 39320, 39322, 39323, 39327, 43177, 43181, 43199, 43244, 43246, 43247, 43262, 43517, 43519, 43688, 43692, 43710, 43720, 43722, 43725, 43726, 43727, 43754, 43756, 43772, 43773, 43967, 43980, 43983, 43997, 44014, 44028, 44030, 44284, 44285, 44286, 44287, 44541, 44543, 44782, 44798, 45052, 45054, 49151, 60139, 60158, 60415, 61155, 61179, 61438, 66047, 66135, 66391, 66399, 66475, 66559, 68351, 68607, 69375, 69631
9	306	2, 23, 26, 30, 43, 61, 62, 111, 137, 138, 139, 143, 169, 172, 173, 235, 283, 383, 393, 395, 426, 430, 447, 491, 494, 559, 575, 583, 590, 591, 602, 606, 639, 650, 651, 655, 683, 687, 706, 710, 714, 746, 751, 859, 904, 905, 906, 907, 911, 936, 937, 942, 966, 967, 968, 969, 971, 973, 988, 989, 1003, 1004, 1005, 1021, 1638, 1655, 1727, 1774, 1790, 1791, 1919, 1928, 1935, 1962, 1963, 1979, 2031, 2032, 2040, 2042, 2043, 2046, 2184, 2216, 2218, 2219, 2235, 2282, 2296, 2298, 2299, 2303, 2555, 2559, 2734, 2751, 2760, 2762, 2763, 2781, 2808, 2812, 2813, 3051, 3054, 3056, 3059, 3070, 3808, 3818, 3825, 3834, 3835, 4082, 4087, 4088, 6079, 6143, 6587, 6655, 6906, 6907, 6910, 7155, 7163, 7920, 7934, 7935, 8176, 8177, 8179, 8186, 8187, 8190, 10922, 10927, 10986, 10990, 10991, 11519, 12014, 12015, 12030, 15871, 16110, 16126, 28398, 32767, 34944, 34947, 34968, 34970, 34971, 34975, 34985, 34989, 34990, 35000, 35001, 35004, 35055, 35065, 35070, 35225, 35241, 35242, 35243, 35259, 35311, 35323, 35499, 35503, 35519, 35528, 35531, 35546, 35549, 35550, 35566, 35567, 35576, 35579, 35580, 35775, 35823, 35835, 36590, 36606, 36607, 36856, 39065, 39099, 39167, 39312, 39313, 39336, 39337, 39339, 39353, 39357, 39359, 39417, 39419, 39645, 39647, 39679, 39899, 39903, 39935, 40959, 43144, 43161, 43182, 43208, 43212, 43245, 43432, 43435, 43437, 43689, 43693, 43708, 43723, 43736, 43737, 43740, 43743, 43755, 43757, 43979, 43996, 43999, 44010, 44011, 44012, 44029, 44236, 44268, 44269, 44270, 44271, 44280, 44525, 44540, 44778, 44780, 44783, 44784, 44785, 44787, 44789, 44794, 44795, 44796, 44797, 45041, 45043, 45048, 45051, 45053, 48383, 48639, 48878, 48894, 48895, 49150, 60074, 60096, 60104, 60106, 60136, 60140, 60394, 60399, 61153, 61158, 61160, 61163, 61170, 61176, 61434, 65256, 65258, 65263, 65963, 66031, 66303, 66441, 66443, 66479, 66511, 66525, 66527, 66543, 66557, 67503, 67583, 68267, 68283, 68303, 68347, 68603, 69359, 69627, 73471, 73727
10	619	6, 8, 9, 25, 44, 45, 106, 110, 142, 152, 154, 155, 159, 188, 189, 190, 279, 298, 299, 301, 302, 317, 367, 392, 394, 399, 409, 411, 424, 428, 429, 490, 510, 554, 558, 574, 578, 579, 586, 597, 605, 648, 654, 666, 667, 671, 680, 702, 703, 707, 711, 712, 715, 716, 717, 727, 730, 733, 734, 748, 764, 858, 862, 879, 908, 909, 910, 921, 923, 927, 940, 941, 962, 964, 965, 970, 981, 983, 985, 986, 987, 990, 1002, 1654, 1663, 1695, 1702, 1706, 1710, 1711, 1782, 1783, 1910, 1930, 1931, 1951, 1955, 1957, 1959, 1960, 1966, 1971, 1975, 1978, 2019, 2023, 2030, 2034, 2185, 2186, 2187, 2191, 2200, 2201, 2217, 2220, 2222, 2223, 2232, 2233, 2234, 2236, 2239, 2286, 2287, 2297, 2302, 2457, 2473, 2474, 2475, 2479, 2489, 2491, 2543, 2553, 2554, 2722, 2723, 2728, 2732, 2736, 2739, 2744, 2750, 2752, 2754, 2755, 2758, 2759, 2768, 2770, 2773, 2775, 2776, 2777, 2778, 2779, 2780, 2784, 2788, 2795, 2796, 2801, 2802, 2803, 2804, 2806, 2807, 2992, 2993, 2995, 3033, 3034, 3035, 3038, 3040, 3041, 3044, 3045, 3050, 3057, 3060, 3061, 3063, 3064, 3068, 3069, 3810, 3826, 3827, 4086, 6031, 6047, 6063, 6296, 6330, 6331, 6394, 6395, 6398, 6399, 6553, 6570, 6585, 6591, 6649, 6651, 6826, 6842, 6843, 6846, 6847, 6874, 6876, 6878, 6879, 6896, 7089, 7091, 7098, 7103, 7129, 7131, 7136, 7140, 7150, 7151, 7152, 7153, 7156, 7157, 7162, 7164, 7165, 7166, 7922, 7930, 8178, 8183, 10410, 10411, 10474, 10475, 10479, 10492, 10493, 10495, 10923, 10926, 10943, 10944, 10956, 10958, 10959, 10965, 10973, 10975, 11004, 11005, 11006, 11199, 11243, 11246, 11247, 11488, 11500, 11502, 11503, 11504, 11516, 11517, 11518, 11745, 11757, 11759, 11760, 11765, 11772, 11773, 11775, 12002, 12010, 12016, 12017, 12018, 12019, 12026, 12027, 12029, 12272, 12273, 12274, 12275, 12277, 12279, 12282, 12283, 12284, 12285, 12286, 15580, 15581, 15599, 15613, 15614, 15837, 15868, 15869, 16124, 16381, 16382, 27328, 27370, 27374, 27391, 28414, 28415, 28671, 32494, 32510, 32511, 32766, 32897, 32898, 32899, 32903, 32904, 32906, 32907, 32910, 32911,

Optimal Cost	Number of Functions	Function Index
		32938, 32942, 32943, 32959, 33023, 33411, 33474, 33475, 33503, 33535, 33664, 33666, 33672, 33673, 33675, 33676, 33731, 33734, 33740, 33743, 33759, 33791, 34945, 34946, 34951, 34958, 34960, 34961, 34963, 34976, 34977, 34978, 34979, 34981, 34992, 34993, 34994, 34995, 35006, 35050, 35056, 35058, 35059, 35063, 35227, 35240, 35247, 35257, 35258, 35263, 35320, 35321, 35322, 35502, 35518, 35544, 35545, 35547, 35548, 35562, 35581, 35770, 35802, 35803, 35822, 35832, 35834, 35836, 35838, 36858, 36859, 36862, 39048, 39056, 39066, 39067, 39071, 39080, 39082, 39096, 39098, 39103, 39160, 39162, 39166, 39315, 39328, 39329, 39341, 39343, 39345, 39352, 39354, 39356, 39416, 39422, 39610, 39611, 39642, 39646, 39867, 39897, 39931, 43145, 43146, 43147, 43148, 43149, 43151, 43160, 43163, 43196, 43197, 43198, 43210, 43213, 43215, 43224, 43228, 43229, 43231, 43240, 43417, 43434, 43436, 43439, 43453, 43468, 43485, 43500, 43501, 43503, 43516, 43709, 43712, 43716, 43721, 43733, 43738, 43739, 43742, 43752, 43964, 43976, 43977, 43981, 43982, 44013, 44239, 44252, 44253, 44256, 44272, 44282, 44283, 44509, 44517, 44524, 44527, 44528, 44533, 44536, 44542, 44768, 44770, 44786, 44791, 45042, 45044, 45047, 48380, 48381, 59626, 59627, 59630, 59631, 59646, 59647, 60075, 60078, 60079, 60095, 60098, 60105, 60107, 60110, 60111, 60117, 60120, 60121, 60122, 60124, 60125, 60127, 60141, 60156, 60157, 60377, 60392, 60396, 60398, 61159, 61174, 61175, 61177, 61431, 61432, 61435, 65823, 65839, 65855, 65929, 65931, 65945, 65961, 65962, 65967, 66111, 66118, 66119, 66134, 66143, 66255, 66271, 66367, 66395, 66431, 66440, 66445, 66447, 66473, 66474, 66495, 66499, 66503, 66505, 66507, 66508, 66509, 67447, 67495, 67499, 67519, 67567, 67571, 67575, 67771, 67833, 67835, 67839, 68257, 68266, 68271, 68282, 68287, 68297, 68298, 68299, 68317, 68319, 68321, 68331, 68334, 68335, 68337, 68341, 68346, 68349, 68539, 68587, 68591, 68595, 69345, 69358, 69361, 69374, 69617, 69619, 69623, 72447, 72635, 72639, 72671, 72699, 72703, 73713, 73723
11	965	24, 40, 126, 128, 129, 130, 131, 135, 232, 281, 282, 286, 297, 300, 316, 318, 385, 387, 398, 408, 410, 415, 444, 445, 489, 555, 576, 587, 596, 603, 604, 622, 623, 642, 643, 647, 649, 652, 653, 664, 665, 681, 684, 704, 705, 708, 709, 725, 726, 728, 731, 732, 747, 765, 830, 856, 857, 876, 878, 892, 893, 894, 896, 898, 899, 903, 920, 922, 924, 925, 956, 958, 961, 980, 984, 1000, 1001, 1634, 1639, 1647, 1650, 1651, 1672, 1674, 1678, 1679, 1689, 1698, 1700, 1701, 1718, 1719, 1726, 1762, 1766, 1770, 1775, 1776, 1778, 1779, 1784, 1786, 1927, 1934, 1945, 1947, 1952, 1953, 1954, 1956, 1958, 1964, 1965, 1968, 1970, 1972, 1973, 1976, 1980, 1982, 2016, 2017, 2018, 2022, 2024, 2026, 2027, 2033, 2038, 2041, 2202, 2203, 2208, 2210, 2211, 2221, 2224, 2227, 2238, 2280, 2283, 2288, 2290, 2291, 2456, 2458, 2459, 2463, 2472, 2478, 2488, 2490, 2492, 2493, 2495, 2539, 2552, 2721, 2729, 2733, 2737, 2738, 2745, 2748, 2753, 2761, 2769, 2771, 2772, 2774, 2785, 2786, 2787, 2789, 2790, 2792, 2797, 2809, 2994, 2997, 2999, 3000, 3001, 3004, 3006, 3024, 3025, 3026, 3027, 3030, 3031, 3032, 3043, 3048, 3052, 3053, 3058, 3062, 3065, 3809, 3811, 3819, 3830, 3831, 3832, 4089, 5886, 5887, 6015, 6041, 6043, 6058, 6059, 6126, 6127, 6280, 6297, 6298, 6314, 6328, 6334, 6335, 6392, 6393, 6552, 6554, 6555, 6559, 6568, 6569, 6571, 6574, 6575, 6584, 6586, 6588, 6589, 6634, 6638, 6648, 6650, 6654, 6816, 6817, 6831, 6832, 6833, 6840, 6858, 6862, 6863, 6864, 6866, 6872, 6873, 6875, 6877, 6880, 6884, 6885, 6890, 6894, 6895, 6897, 6898, 6899, 6901, 6904, 6905, 6908, 6909, 7088, 7093, 7097, 7101, 7120, 7121, 7123, 7127, 7130, 7134, 7141, 7146, 7147, 7148, 7149, 7154, 7158, 7159, 7160, 7161, 7904, 7918, 7921, 7923, 7931, 8184, 8185, 10376, 10408, 10414, 10415, 10430, 10431, 10444, 10472, 10476, 10478, 10494, 10749, 10751, 10942, 10948, 10949, 10952, 10954, 10955, 10957, 10972, 10987, 10988, 11200, 11210, 11212, 11214, 11215, 11220, 11228, 11229, 11230, 11231, 11242, 11244, 11260, 11261, 11262, 11468, 11501, 11508, 11509, 11512, 11744, 11749, 11756, 11761, 11768, 12000, 12003, 12012, 12021, 12023, 12028, 12276, 12278, 12280, 15560, 15561, 15565, 15573, 15575, 15583, 15594, 15595, 15596, 15597, 15598, 15829, 15836, 15852, 15853, 15855, 16106, 16111, 16125, 26879, 27135, 27375, 27390, 27627, 27647, 28394, 28399, 28410, 28662, 32902, 32905, 32920, 32921, 32922, 32923, 32927, 32936, 32939, 32940, 32941, 32956, 32958, 33002, 33006, 33007, 33022, 33152, 33154, 33155, 33160, 33161, 33162, 33163, 33167, 33193, 33194, 33195, 33199, 33215, 33263, 33279, 33408, 33413, 33414, 33415, 33416, 33418, 33419, 33420, 33421, 33422, 33423, 33430, 33450, 33454, 33455, 33470, 33471, 33472, 33478, 33479, 33484, 33486, 33487, 33495, 33501, 33502, 33518, 33534, 33665, 33670, 33671, 33674, 33677, 33679, 33687, 33706, 33707, 33711, 33727, 33728, 33730, 33735, 33736, 33737, 33741, 33742, 33751, 33757, 33775, 34551, 34559, 34688, 34690, 34691, 34696, 34698, 34699, 34703, 34710, 34711, 34720, 34724, 34725, 34726, 34727, 34730, 34735, 34740, 34741, 34743, 34751, 34800, 34803, 34807, 34815, 34950, 34980, 34982, 34983, 34996, 34997, 34999, 35005, 35040, 35042, 35043, 35048, 35051, 35057, 35217, 35224, 35226, 35231, 35232, 35233, 35235, 35237, 35244, 35245, 35246, 35249, 35251, 35256, 35260, 35261, 35305, 35307, 35310, 35312, 35313, 35315, 35326, 35488, 35490, 35491, 35493, 35496, 35500, 35504, 35505, 35506, 35507, 35512, 35513, 35516, 35520, 35522, 35523, 35529, 35536, 35538, 35539, 35541, 35543, 35552, 35563, 35564, 35568, 35569, 35570, 35571, 35572, 35573, 35575, 35577, 35760, 35763, 35768, 35772, 35792, 35794, 35795, 35799, 35800, 35801, 35806, 35818, 35819, 35820, 35824, 35826, 35827, 35828, 35829, 35831, 35833, 35837, 36591, 36592, 36602, 36603, 36848, 36850, 36851, 36855, 36857, 39049, 39050, 39051, 39055, 39057, 39058, 39059, 39070, 39081, 39083, 39086, 39087, 39097, 39100, 39101, 39102, 39161, 39163, 39314, 39318, 39319, 39326,

Optimal Cost	Number of Functions	Function Index
		39330, 39331, 39332, 39333, 39334, 39335, 39340, 39342, 39344, 39346, 39347, 39348, 39349, 39351, 39358, 39407, 39408, 39409, 39410, 39411, 39414, 39415, 39418, 39594, 39608, 39615, 39626, 39630, 39631, 39640, 39641, 39643, 39644, 39674, 39678, 39864, 39865, 39866, 39868, 39871, 39902, 39918, 39919, 39929, 39932, 40703, 43150, 43164, 43165, 43167, 43209, 43211, 43214, 43225, 43226, 43230, 43242, 43243, 43416, 43418, 43419, 43420, 43421, 43423, 43452, 43455, 43465, 43469, 43471, 43481, 43487, 43497, 43502, 43518, 43713, 43714, 43715, 43717, 43719, 43732, 43735, 43965, 43966, 43968, 43971, 43972, 43974, 43975, 43978, 43988, 43989, 43991, 43992, 43993, 43994, 43995, 43998, 44008, 44232, 44234, 44237, 44238, 44255, 44259, 44264, 44266, 44273, 44275, 44276, 44277, 44279, 44281, 44506, 44508, 44511, 44512, 44513, 44519, 44521, 44526, 44529, 44531, 44532, 44535, 44537, 44538, 44539, 44769, 44771, 44772, 44773, 44774, 44776, 44779, 44781, 44788, 44790, 44792, 45046, 45049, 48332, 48348, 48349, 48364, 48366, 48367, 48382, 48605, 48620, 48636, 48637, 48879, 48892, 49148, 49149, 59560, 59562, 59625, 59883, 59887, 59903, 60094, 60097, 60099, 60102, 60103, 60118, 60119, 60123, 60126, 60137, 60350, 60351, 60375, 60376, 60379, 60383, 60393, 60397, 60412, 60413, 60414, 61161, 61430, 61433, 65259, 65817, 65819, 65835, 65853, 65919, 65935, 65960, 65965, 65966, 65983, 66025, 66027, 66030, 66046, 66094, 66095, 66113, 66115, 66127, 66133, 66138, 66139, 66175, 66241, 66243, 66246, 66247, 66252, 66253, 66263, 66269, 66301, 66390, 66394, 66442, 66444, 66457, 66459, 66463, 66472, 66477, 66478, 66496, 66497, 66500, 66501, 66504, 66510, 66517, 66519, 66523, 66524, 66539, 66540, 66541, 66542, 66556, 66558, 67191, 67215, 67247, 67311, 67319, 67327, 67455, 67471, 67493, 67507, 67511, 67515, 67569, 67579, 67737, 67769, 67775, 68009, 68011, 68027, 68091, 68095, 68259, 68265, 68269, 68270, 68273, 68281, 68289, 68291, 68295, 68296, 68302, 68305, 68309, 68313, 68315, 68316, 68320, 68325, 68330, 68336, 68339, 68340, 68343, 68345, 68348, 68350, 68529, 68531, 68543, 68571, 68575, 68577, 68592, 68593, 68596, 68597, 68602, 68604, 68605, 68606, 69344, 69355, 69360, 69363, 69370, 69371, 69616, 69618, 69626, 69630, 71615, 71679, 72107, 72123, 72187, 72191, 72379, 72383, 72415, 72443, 72625, 72667, 72687, 72689, 73457, 73715, 75947, 76029, 76031, 76459, 76463, 76479, 76495, 76523, 76526, 76527, 76543, 76779, 76783, 76799, 77037, 77039, 77053, 77055, 77281
12	1237	22, 41, 104, 105, 107, 134, 150, 158, 233, 280, 296, 361, 362, 363, 366, 384, 386, 391, 446, 488, 556, 557, 572, 573, 577, 584, 585, 600, 601, 618, 620, 636, 638, 640, 645, 646, 662, 668, 669, 670, 685, 700, 713, 724, 729, 744, 745, 749, 829, 872, 874, 875, 897, 900, 901, 902, 918, 919, 926, 957, 982, 1632, 1635, 1642, 1646, 1658, 1662, 1666, 1670, 1673, 1675, 1686, 1687, 1690, 1691, 1694, 1696, 1699, 1703, 1704, 1705, 1707, 1708, 1709, 1712, 1715, 1716, 1717, 1721, 1722, 1723, 1724, 1725, 1760, 1763, 1767, 1768, 1777, 1785, 1787, 1912, 1914, 1915, 1918, 1920, 1922, 1923, 1926, 1929, 1939, 1943, 1944, 1946, 1961, 1969, 1974, 1977, 1981, 2025, 2176, 2179, 2190, 2207, 2209, 2212, 2213, 2214, 2215, 2225, 2226, 2228, 2237, 2272, 2273, 2274, 2281, 2289, 2295, 2448, 2449, 2464, 2465, 2466, 2467, 2469, 2476, 2477, 2480, 2481, 2482, 2483, 2485, 2494, 2529, 2530, 2531, 2536, 2537, 2538, 2542, 2544, 2545, 2546, 2547, 2550, 2551, 2558, 2724, 2726, 2727, 2740, 2741, 2743, 2749, 2791, 2793, 2996, 3005, 3042, 3046, 3047, 3049, 3814, 3815, 3816, 3833, 5759, 5786, 5790, 5791, 5802, 5806, 5820, 5822, 5823, 5866, 5870, 6023, 6024, 6025, 6026, 6027, 6030, 6039, 6042, 6046, 6056, 6060, 6061, 6062, 6076, 6077, 6078, 6120, 6122, 6123, 6142, 6281, 6299, 6303, 6304, 6312, 6315, 6318, 6319, 6320, 6323, 6329, 6332, 6368, 6370, 6378, 6382, 6384, 6387, 6544, 6545, 6560, 6561, 6562, 6572, 6573, 6576, 6577, 6579, 6590, 6624, 6626, 6630, 6633, 6639, 6640, 6641, 6643, 6647, 6818, 6819, 6821, 6824, 6825, 6827, 6830, 6834, 6835, 6836, 6837, 6839, 6841, 6848, 6850, 6851, 6856, 6859, 6865, 6867, 6869, 6881, 6882, 6888, 6891, 6892, 6893, 6900, 6903, 7090, 7092, 7095, 7096, 7100, 7102, 7122, 7126, 7128, 7137, 7138, 7139, 7142, 7143, 7144, 7145, 7905, 7906, 7907, 7914, 7915, 7919, 7926, 7927, 7928, 8182, 10380, 10393, 10396, 10409, 10412, 10413, 10428, 10432, 10435, 10440, 10441, 10452, 10453, 10456, 10460, 10461, 10477, 10649, 10665, 10667, 10669, 10671, 10697, 10700, 10717, 10729, 10731, 10733, 10735, 10748, 10920, 10924, 10925, 10945, 10947, 10964, 10967, 10968, 10969, 10970, 10971, 10974, 10984, 10989, 11203, 11204, 11208, 11209, 11211, 11213, 11221, 11223, 11224, 11225, 11226, 11245, 11456, 11458, 11459, 11469, 11471, 11472, 11473, 11475, 11484, 11485, 11487, 11489, 11490, 11491, 11492, 11493, 11496, 11498, 11499, 11505, 11506, 11507, 11510, 11511, 11513, 11514, 11515, 11728, 11741, 11747, 11751, 11752, 11753, 11758, 11762, 11763, 11764, 11767, 11769, 11770, 11771, 11774, 12001, 12004, 12006, 12007, 12008, 12011, 12013, 12020, 12022, 12024, 12025, 12281, 15553, 15554, 15556, 15557, 15558, 15559, 15562, 15563, 15566, 15572, 15576, 15577, 15579, 15582, 15592, 15831, 15833, 15839, 15850, 15854, 15870, 16107, 16108, 16109, 26856, 27115, 27306, 27336, 27338, 27342, 27343, 27349, 27371, 27630, 27631, 28384, 28386, 28387, 28390, 28402, 28403, 28406, 28411, 28663, 28670, 32490, 32495, 32768, 32770, 32771, 32776, 32778, 32779, 32782, 32783, 32810, 32814, 32815, 32831, 32918, 32919, 32937, 32957, 33003, 33159, 33166, 33177, 33179, 33183, 33192, 33197, 33198, 33258, 33259, 33262, 33409, 33412, 33417, 33431, 33433, 33434, 33435, 33438, 33439, 33451, 33473, 33476, 33477, 33480, 33481, 33482, 33483, 33485, 33493, 33494, 33496, 33497, 33498, 33499, 33500, 33514, 33515, 33516, 33519, 33532, 33668, 33669, 33678, 33688, 33689, 33691, 33692, 33693, 33695, 33704, 33708, 33710, 33724, 33726, 33729, 33732, 33733, 33738, 33739, 33748, 33749, 33750, 33752, 33753, 33754, 33755, 33756, 33758, 33770, 33771, 33772, 33774, 33788, 33789, 33790, 34432, 34433,

Optimal Cost	Number of Functions	Function Index
		34434, 34435, 34439, 34440, 34442, 34446, 34447, 34454, 34455, 34468, 34469, 34470, 34471, 34478, 34479, 34484, 34485, 34486, 34487, 34494, 34495, 34528, 34534, 34542, 34543, 34544, 34545, 34547, 34550, 34558, 34689, 34694, 34697, 34706, 34707, 34721, 34722, 34723, 34728, 34729, 34731, 34732, 34733, 34734, 34736, 34737, 34739, 34742, 34744, 34747, 34748, 34749, 34750, 34784, 34785, 34786, 34787, 34791, 34795, 34799, 34801, 34802, 34808, 34810, 34811, 34962, 34966, 34967, 34974, 34998, 35041, 35046, 35047, 35049, 35062, 35216, 35219, 35234, 35236, 35238, 35239, 35248, 35250, 35252, 35253, 35255, 35262, 35296, 35306, 35314, 35318, 35319, 35489, 35492, 35495, 35497, 35501, 35508, 35509, 35511, 35517, 35521, 35526, 35527, 35537, 35540, 35542, 35553, 35554, 35555, 35556, 35557, 35560, 35565, 35574, 35761, 35762, 35764, 35769, 35773, 35774, 35793, 35808, 35809, 35811, 35813, 35815, 35816, 35817, 35821, 35825, 35830, 36576, 36584, 36586, 36587, 36593, 36594, 36595, 36599, 36600, 36601, 36849, 38530, 38531, 38534, 38535, 38551, 38552, 38553, 38554, 38555, 38558, 38559, 38570, 38575, 38590, 38591, 38654, 38655, 38786, 38787, 38790, 38791, 38806, 38809, 38810, 38811, 38815, 38826, 38827, 38831, 38847, 38911, 39040, 39042, 39043, 39062, 39072, 39073, 39074, 39075, 39076, 39077, 39078, 39084, 39085, 39088, 39089, 39090, 39091, 39092, 39093, 39095, 39146, 39150, 39151, 39152, 39153, 39154, 39155, 39159, 39350, 39392, 39393, 39394, 39395, 39399, 39400, 39401, 39402, 39403, 39584, 39589, 39590, 39592, 39593, 39595, 39597, 39599, 39600, 39601, 39602, 39603, 39605, 39606, 39607, 39609, 39612, 39613, 39614, 39616, 39617, 39619, 39624, 39625, 39627, 39632, 39633, 39634, 39635, 39636, 39637, 39638, 39639, 39662, 39663, 39664, 39665, 39669, 39671, 39672, 39673, 39675, 39676, 39677, 39857, 39859, 39861, 39863, 39869, 39870, 39888, 39889, 39890, 39891, 39895, 39896, 39898, 39912, 39914, 39915, 39916, 39917, 39920, 39921, 39923, 39924, 39925, 39926, 39927, 39928, 39930, 39933, 39934, 40686, 40696, 40702, 40944, 40945, 40946, 40947, 40950, 40951, 40952, 40953, 40954, 40955, 40958, 43136, 43139, 43140, 43141, 43143, 43156, 43157, 43162, 43166, 43200, 43203, 43204, 43205, 43220, 43221, 43227, 43241, 43438, 43456, 43457, 43464, 43467, 43470, 43480, 43482, 43483, 43484, 43486, 43496, 43499, 43718, 43734, 43753, 43969, 43970, 43973, 44009, 44224, 44233, 44235, 44240, 44241, 44245, 44248, 44250, 44254, 44257, 44258, 44260, 44261, 44267, 44274, 44278, 44504, 44505, 44507, 44510, 44515, 44516, 44520, 44522, 44523, 44530, 44534, 44775, 44793, 48328, 48330, 48331, 48333, 48335, 48351, 48360, 48361, 48362, 48363, 48365, 48601, 48602, 48603, 48604, 48607, 48616, 48621, 48622, 48623, 48638, 48874, 48875, 48876, 48877, 48893, 59528, 59535, 59545, 59561, 59563, 59564, 59565, 59566, 59567, 59581, 59583, 59801, 59817, 59821, 59880, 59882, 59886, 60072, 60076, 60077, 60116, 60378, 65257, 65834, 65837, 65838, 65852, 65903, 65921, 65923, 65928, 65930, 65947, 65951, 65964, 65981, 66026, 66090, 66091, 66093, 66109, 66110, 66114, 66121, 66122, 66123, 66126, 66132, 66137, 66141, 66142, 66159, 66240, 66244, 66245, 66261, 66265, 66267, 66270, 66300, 66365, 66393, 66398, 66415, 66429, 66432, 66433, 66434, 66435, 66439, 66446, 66458, 66461, 66476, 66498, 66502, 66506, 66520, 66521, 66522, 66526, 66536, 66537, 66538, 67175, 67183, 67199, 67209, 67211, 67231, 67239, 67241, 67242, 67243, 67245, 67246, 67255, 67259, 67263, 67303, 67307, 67310, 67315, 67318, 67322, 67323, 67326, 67451, 67465, 67467, 67487, 67489, 67491, 67492, 67496, 67497, 67498, 67501, 67509, 67555, 67559, 67562, 67563, 67566, 67568, 67576, 67577, 67578, 67739, 67761, 67763, 67773, 67824, 67825, 67827, 67993, 67995, 68001, 68008, 68010, 68013, 68015, 68025, 68031, 68073, 68075, 68079, 68089, 68090, 68256, 68258, 68261, 68264, 68268, 68272, 68275, 68277, 68279, 68280, 68285, 68288, 68290, 68304, 68307, 68311, 68312, 68314, 68318, 68323, 68324, 68329, 68332, 68333, 68338, 68342, 68344, 68528, 68536, 68537, 68538, 68563, 68569, 68570, 68579, 68581, 68586, 68588, 68590, 68599, 68600, 68601, 69346, 69347, 69362, 69622, 69624, 69625, 71423, 71551, 71577, 71583, 71595, 71599, 71663, 71865, 71867, 71929, 71931, 71935, 72089, 72091, 72105, 72121, 72127, 72185, 72353, 72361, 72362, 72363, 72367, 72369, 72377, 72378, 72393, 72395, 72399, 72411, 72413, 72414, 72417, 72427, 72431, 72433, 72435, 72437, 72442, 72446, 72627, 72633, 72657, 72659, 72665, 72683, 72686, 72691, 72693, 72695, 72697, 72698, 72701, 73440, 73441, 73443, 73451, 73455, 73456, 73459, 73466, 73467, 73470, 73712, 73719, 73722, 73726, 75945, 76011, 76458, 76462, 76481, 76492, 76493, 76509, 76511, 76522, 76525, 76541, 76542, 76735, 77025, 77036, 77040, 77041, 77052, 467985435
13	1004	151, 278, 360, 382, 390, 407, 414, 552, 619, 637, 641, 644, 660, 661, 663, 701, 873, 877, 916, 917, 1656, 1659, 1664, 1665, 1667, 1671, 1680, 1681, 1682, 1683, 1688, 1697, 1713, 1714, 1720, 1761, 1771, 1921, 1936, 1937, 1938, 1942, 1950, 2177, 2178, 2183, 2192, 2193, 2194, 2195, 2206, 2229, 2230, 2231, 2275, 2278, 2279, 2294, 2451, 2468, 2470, 2471, 2484, 2487, 2528, 2534, 2535, 2742, 5762, 5767, 5770, 5774, 5775, 5782, 5784, 5785, 5787, 5804, 5805, 5807, 5821, 5871, 6016, 6017, 6018, 6019, 6022, 6038, 6040, 6057, 6272, 6273, 6275, 6282, 6283, 6287, 6288, 6289, 6290, 6291, 6302, 6305, 6306, 6307, 6308, 6313, 6316, 6317, 6321, 6322, 6324, 6325, 6333, 6371, 6374, 6375, 6376, 6379, 6383, 6385, 6386, 6390, 6391, 6547, 6563, 6564, 6565, 6566, 6567, 6578, 6580, 6581, 6583, 6625, 6627, 6631, 6632, 6635, 6642, 6646, 6820, 6822, 6828, 6829, 6838, 6844, 6845, 6849, 6854, 6855, 6857, 6868, 6870, 6871, 6883, 6886, 6887, 6889, 6902, 7910, 7911, 7929, 10368, 10370, 10371, 10373, 10377, 10378, 10379, 10381, 10382, 10383, 10392, 10397, 10429, 10433, 10434, 10436, 10437, 10438, 10439, 10442, 10443, 10445, 10446, 10447, 10455, 10457, 10458, 10459, 10462, 10463, 10473, 10664, 10666, 10668, 10670, 10684, 10685, 10687, 10688, 10689,

Optimal Cost	Number of Functions	Function Index
		10691, 10696, 10701, 10709, 10712, 10713, 10716, 10730, 10732, 10734, 10750, 10921, 10940, 10941, 10946, 10950, 10951, 10953, 10966, 10985, 11196, 11197, 11198, 11201, 11202, 11205, 11207, 11227, 11240, 11241, 11457, 11460, 11462, 11464, 11465, 11466, 11470, 11474, 11476, 11477, 11479, 11480, 11481, 11486, 11494, 11495, 11497, 11729, 11730, 11731, 11733, 11735, 11736, 11738, 11740, 11742, 11743, 11746, 11748, 11750, 11754, 11755, 11766, 12005, 15574, 15578, 15593, 15828, 15832, 15834, 15835, 15838, 15848, 15849, 15851, 16104, 26760, 26776, 26777, 26792, 26794, 26797, 26858, 26859, 26862, 26863, 26878, 27033, 27048, 27049, 27050, 27051, 27053, 27055, 27113, 27114, 27118, 27119, 27305, 27307, 27310, 27311, 27327, 27330, 27331, 27334, 27335, 27339, 27352, 27353, 27354, 27355, 27356, 27357, 27358, 27359, 27368, 27369, 27372, 27373, 27388, 27389, 27607, 27609, 27611, 27614, 27615, 27626, 27628, 27629, 27644, 27646, 28391, 28392, 28395, 28407, 28408, 28409, 28664, 28665, 28666, 28667, 32769, 32775, 32777, 32792, 32793, 32794, 32795, 32798, 32799, 32808, 32811, 32812, 32813, 32828, 32829, 32830, 32874, 32878, 32895, 32926, 33000, 33070, 33071, 33087, 33158, 33175, 33176, 33178, 33196, 33212, 33213, 33214, 33278, 33320, 33324, 33325, 33327, 33340, 33342, 33343, 33344, 33345, 33346, 33347, 33350, 33351, 33354, 33355, 33358, 33359, 33365, 33366, 33367, 33370, 33371, 33372, 33373, 33374, 33375, 33405, 33407, 33428, 33429, 33432, 33436, 33437, 33448, 33452, 33453, 33468, 33492, 33517, 33533, 33596, 33599, 33622, 33623, 33626, 33631, 33660, 33663, 33684, 33685, 33686, 33690, 33694, 33705, 33709, 33725, 33773, 34406, 34423, 34441, 34443, 34448, 34449, 34450, 34451, 34457, 34458, 34462, 34463, 34464, 34465, 34466, 34467, 34474, 34475, 34480, 34481, 34482, 34483, 34491, 34530, 34531, 34546, 34679, 34702, 34704, 34705, 34712, 34713, 34714, 34715, 34718, 34719, 34738, 34745, 34746, 34790, 34794, 34798, 34806, 34809, 34814, 35218, 35223, 35254, 35297, 35298, 35299, 35302, 35303, 35304, 35494, 35510, 35558, 35559, 35561, 35765, 35767, 35798, 35810, 35812, 35814, 36577, 36578, 36579, 36582, 36585, 36598, 36854, 38528, 38536, 38538, 38542, 38543, 38574, 38784, 38792, 38794, 38795, 38799, 38808, 38814, 38829, 38830, 38844, 38845, 38846, 38895, 38910, 39041, 39054, 39063, 39079, 39094, 39138, 39144, 39147, 39158, 39398, 39585, 39586, 39588, 39591, 39598, 39604, 39618, 39622, 39623, 39648, 39653, 39654, 39656, 39658, 39660, 39661, 39666, 39667, 39668, 39670, 39856, 39858, 39860, 39894, 39904, 39905, 39906, 39907, 39908, 39909, 39910, 39911, 39913, 39922, 40687, 40688, 40691, 40694, 40695, 40697, 40698, 40699, 43137, 43138, 43142, 43159, 43201, 43202, 43206, 43207, 43223, 43413, 43454, 43459, 43460, 43461, 43463, 43466, 43476, 43477, 43498, 44225, 44226, 44227, 44228, 44229, 44231, 44243, 44244, 44247, 44249, 44251, 44262, 44263, 44265, 44496, 44497, 44498, 44500, 44501, 44503, 44514, 44518, 44777, 48320, 48323, 48324, 48325, 48327, 48329, 48334, 48340, 48341, 48344, 48345, 48346, 48347, 48350, 48597, 48600, 48606, 48617, 48618, 48619, 48872, 59529, 59530, 59531, 59534, 59544, 59547, 59551, 59580, 59582, 59816, 59818, 59819, 59820, 59822, 59823, 59836, 59837, 59839, 59902, 60073, 60092, 60093, 60348, 60349, 60374, 60382, 65815, 65818, 65822, 65832, 65833, 65836, 65854, 65899, 65902, 65920, 65922, 65927, 65934, 65943, 65944, 65946, 65980, 65982, 66024, 66092, 66108, 66112, 66120, 66136, 66140, 66154, 66158, 66174, 66264, 66268, 66364, 66366, 66392, 66411, 66414, 66428, 66430, 66437, 66438, 66453, 66455, 66456, 66460*, 66492, 66493, 66494, 66516*, 66518, 67169, 67171, 67174, 67187, 67190, 67201, 67203, 67206, 67207*, 67223, 67225, 67233, 67235, 67237, 67238, 67240*, 67251, 67253*, 67254, 67257, 67297, 67299, 67302, 67305, 67306, 67312, 67313, 67314, 67321, 67457, 67459, 67463, 67464, 67466, 67479, 67481, 67483, 67488, 67490, 67494, 67500, 67502, 67504, 67505, 67508, 67510, 67512, 67513, 67514, 67516, 67517, 67552*, 67553, 67561, 67570, 67574, 67582, 67729, 67743, 67760, 67762, 67765, 67826, 67830, 67831, 67985, 67999, 68000, 68003, 68005, 68012, 68017, 68019, 68024, 68026, 68029, 68074, 68078, 68080, 68081, 68082, 68083, 68088, 68260, 68263, 68274, 68276, 68284, 68286, 68294, 68306, 68308, 68310*, 68322, 68326*, 68327, 68328*, 68530, 68533, 68535, 68541, 68561, 68562, 68567, 68568, 68574, 68576, 68580, 68583*, 68585, 68589, 68594, 68598, 69351, 69353, 69354, 69366, 69367, 69368, 69369, 71359, 71563, 71567, 71575, 71579, 71593, 71613, 71659, 71817, 71819, 71823, 71832, 71833, 71835, 71843, 71848, 71849, 71850*, 71851, 71853, 71855, 71857, 71864, 71866, 71869*, 71871, 71907, 71915, 71921, 71923, 71928, 72095, 72097, 72104, 72106, 72109, 72111, 72113, 72115, 72120, 72122, 72125, 72171, 72174, 72175, 72179, 72186, 72352, 72357, 72368, 72371, 72376*, 72382, 72392*, 72394*, 72401, 72409, 72410, 72412*, 72416, 72419, 72421, 72423, 72426, 72429, 72430, 72432, 72436, 72441, 72445, 72624, 72629, 72631, 72634, 72637, 72663, 72666, 72670, 72672, 72673, 72675, 72677, 72679, 72681*, 72682, 72685, 72688, 72692, 72702*, 73450, 73454, 73458, 73463, 73465, 73714, 73721, 75913, 75929, 75944, 75946, 75948*, 75949, 75951, 75967, 75969, 75977*, 75981, 75989, 75997, 76008*, 76009, 76012, 76013, 76015, 76028, 76201, 76203, 76265, 76269, 76271, 76285, 76287, 76457, 76460, 76461, 76478*, 76483, 76485, 76488*, 76489, 76491, 76501, 76503, 76505, 76508, 76540, 76745*, 76747, 76748, 76749, 76751, 76757, 76759, 76765, 76767, 76782, 76797, 76993*, 76995, 77004, 77005, 77007, 77009, 77020, 77021, 77024, 77027*, 77033*, 77038, 77043, 77045*, 77049, 77054, 77265*, 77277, 77279*, 77280, 77283, 77285
14	601	406, 553, 616, 617, 621, 1633, 1640, 1641, 1643, 1657, 1769, 1913, 2182, 2198, 2199, 2450, 2454, 2455, 2462, 2486, 2998, 3817, 5738, 5742, 5743, 5758, 5760, 5763, 5766, 5768, 5769, 5771, 5783, 5800, 5801, 5803, 5864, 5865, 5867, 6014, 6121, 6274, 6286, 6309, 6310, 6311, 6326, 6327, 6369,

Optimal Cost	Number of Functions	Function Index
		6377, 6546, 6550, 6551, 6558, 6582, 6823, 7094, 7912, 7913, 10369, 10372, 10374, 10375, 10388, 10389, 10390, 10391, 10394, 10395, 10398, 10399, 10454, 10644, 10645, 10646, 10648, 10651, 10652, 10653, 10655, 10686, 10690, 10692, 10693, 10694, 10695, 10698, 10699, 10702, 10703, 10708, 10710, 10711, 10714, 10715, 10718, 10719, 10728, 11206, 11222, 11461, 11463, 11467, 11478, 11482, 11483, 11732, 11734, 11737, 11739, 12009, 15830, 16105, 26752, 26754, 26758, 26762, 26766, 26767, 26775, 26778, 26793, 26795, 26796, 26798, 26799, 26812, 26814, 26815, 26857, 27030, 27034, 27035, 27039, 27054, 27068, 27069, 27070, 27071, 27112, 27134, 27304, 27308, 27309, 27325, 27326, 27329, 27337, 27348, 27350, 27351, 27582, 27583, 27606, 27608, 27610, 27624, 27625, 27645, 28385, 32488, 32491, 32774, 32790, 32791, 32809, 32872, 32875, 32879, 32894, 33001, 33047, 33048, 33049, 33050, 33051, 33054, 33055, 33064, 33065, 33066, 33067, 33068, 33069, 33084, 33085, 33086, 33134, 33135, 33150, 33151, 33174, 33182, 33256, 33257, 33322, 33323, 33326, 33341, 33352, 33353, 33364, 33368, 33369, 33384, 33385, 33388, 33390, 33391, 33404, 33406, 33449, 33469, 33512, 33597, 33598, 33624, 33625, 33627, 33630, 33640, 33641, 33643, 33644, 33646, 33647, 33661, 33662, 33768, 33769, 34403, 34408, 34415, 34418, 34419, 34422, 34425, 34427, 34456, 34459, 34472, 34473, 34476, 34477, 34488, 34490, 34492, 34493, 34529, 34535, 34538, 34539, 34552, 34553, 34554, 34555, 34678, 34680, 34682, 34683, 34687, 34792, 34793, 35222, 35230, 35766, 36583, 38529, 38537, 38539, 38568, 38569, 38571, 38572, 38573, 38588, 38589, 38635, 38638, 38639, 38785, 38793, 38798, 38824, 38825, 38828, 38890, 38891, 38894, 39046, 39047, 39136, 39137, 39139, 39142, 39143, 39145, 39587, 39596, 39649, 39650, 39651, 39652, 39655, 39657, 39659, 39862, 40679, 40681, 40682, 40683, 40689, 40690, 43158, 43222, 43412, 43414, 43415, 43422, 43458, 43462, 43478, 43479, 43990, 44230, 44242, 44246, 44499, 44502, 48321, 48322, 48326, 48342, 48343, 48596, 48598, 48599, 48873, 59520, 59546, 59550, 59800, 59802, 59803, 59806, 59807, 59838, 65816, 65897, 65898, 65918, 66088, 66089, 66153, 66155, 66172*, 66173*, 66260*, 66262*, 66410*, 66412*, 66436*, 66462*, 67170*, 67182*, 67186*, 67198*, 67200*, 67222*, 67227*, 67232*, 67234*, 67236*, 67244*, 67252*, 67258*, 67262*, 67296*, 67298*, 67320*, 67446*, 67450*, 67456*, 67458*, 67462*, 67470*, 67482*, 67506*, 67518*, 67554*, 67558*, 67560*, 67731*, 67986*, 67987*, 67994*, 67998*, 68002*, 68014*, 68016*, 68018*, 68021*, 68030*, 68064*, 68066*, 68067*, 68070*, 68072*, 68086*, 68087*, 68094*, 68262*, 68532*, 68540*, 68542*, 68560*, 68566*, 68578*, 68582*, 68584*, 69350*, 71295*, 71321*, 71327*, 71341*, 71343*, 71357*, 71401*, 71403*, 71407*, 71422*, 71561*, 71594*, 71597*, 71598*, 71656*, 71657*, 71658*, 71662*, 71678*, 71816*, 71834*, 71839*, 71854*, 71856*, 71859*, 71868*, 71870*, 71905*, 71911*, 71919*, 71920*, 71930*, 71934*, 72088*, 72099*, 72110*, 72112*, 72124*, 72126*, 72162*, 72163*, 72166*, 72170*, 72176*, 72177*, 72178*, 72183*, 72184*, 72190*, 72355*, 72372*, 72373*, 72375*, 72381*, 72385*, 72387*, 72391*, 72398*, 72400*, 72402*, 72403*, 72405*, 72407*, 72408*, 72418*, 72420*, 72424*, 72425*, 72428*, 72434*, 72439*, 72440*, 72444*, 72626*, 72628*, 72632*, 72636*, 72638*, 72656*, 72658*, 72662*, 72674*, 72676*, 72678*, 72680*, 72684*, 72690*, 72694*, 72696*, 72700*, 73442*, 73447*, 73449*, 73464*, 73718*, 73720*, 75907*, 75912*, 75915*, 75916*, 75933*, 75950*, 75964*, 75965*, 75971*, 75976*, 75979*, 75980*, 75983*, 75991*, 75993*, 75996*, 76010*, 76014*, 76030*, 76205*, 76207*, 76221*, 76223*, 76233*, 76237*, 76249*, 76253*, 76267*, 76268*, 76456*, 76477*, 76480*, 76482*, 76484*, 76487*, 76490*, 76494*, 76500*, 76504*, 76507*, 76510*, 76521*, 76524*, 76733*, 76736*, 76737*, 76739*, 76740*, 76741*, 76743*, 76744*, 76746*, 76750*, 76756*, 76758*, 76761*, 76763*, 76764*, 76766*, 76777*, 76778*, 76780*, 76781*, 76796*, 76798*, 77011*, 77013*, 77023*, 77026*, 77029*, 77035*, 77042*, 77044*, 77047*, 77048*, 77050*, 77051*, 77264*, 77269*, 77276*, 77282*, 401139735, 467199015, 518119710
15	230	5736, 5737, 5739, 5761*, 6278*, 6279*, 6294*, 6295*, 10647, 10650, 10654, 26755, 26759, 26761, 26763, 26774, 26779, 26782, 26783, 26813, 27031, 27032, 27038, 27052, 27324, 27580, 27581, 28393, 32489, 32873, 33131, 33321, 33386, 33387, 33389, 33513, 33642, 33645, 34400, 34401, 34402, 34407, 34409, 34410, 34411, 34414, 34424, 34426, 34430, 34431, 34489, 34536, 34537, 34681, 34686, 38504, 38505, 38506, 38507, 38510, 38511, 38526, 38527, 38632, 38633, 38634, 38783, 38888, 38889, 40672, 40673, 40674, 40675, 40678, 40680, 59521, 59522, 59523, 59526, 59527, 59543, 65814*, 65926*, 65950*, 66152*, 66156*, 66157*, 66408*, 66409*, 66413*, 66454*, 67168*, 67177*, 67178*, 67179*, 67192*, 67193*, 67194*, 67195*, 67202*, 67216*, 67217*, 67219*, 67224*, 67226*, 67230*, 67248*, 67249*, 67250*, 67256*, 67260*, 67261*, 67304*, 67448*, 67449*, 67454*, 67472*, 67473*, 67475*, 67478*, 67480*, 67486*, 67728*, 67730*, 67767*, 67984*, 67992*, 68006*, 68007*, 68020*, 68023*, 68028*, 68065*, 68071*, 68278*, 68534*, 69352*, 71279*, 71305*, 71307*, 71311*, 71319*, 71323*, 71337*, 71338*, 71339*, 71400*, 71402*, 71406*, 71555*, 71559*, 71560*, 71562*, 71576*, 71578*, 71592*, 71596*, 71612*, 71614*, 71808*, 71818*, 71840*, 71841*, 71852*, 71904*, 71906*, 71912*, 71914*, 71918*, 71922*, 71927*, 72080*, 72081*, 72083*, 72096*, 72098*, 72114*, 72117*, 72119*, 72161*, 72167*, 72168*, 72360*, 72365*, 72366*, 72370*, 72384*, 72404*, 72664*, 75904*, 75905*, 75906*, 75914*, 75917*, 75919*, 75928*, 75966*, 75968*, 75982*, 75999*, 76185*, 76200*, 76202*, 76239*, 76245*, 76247*, 76255*, 76284*, 76476*, 76506*, 76520*, 76734*, 76760*, 76762*, 76776*, 76992*, 76994*, 76997*, 77001*, 77006*, 77008*, 77017*,

Optimal Cost	Number of Functions	Function Index
		77028*, 77031*, 77032*, 77034*, 77266*, 77267*, 77268*, 77278*
16	95	26753*, 33046*, 33128*, 33129*, 33130*, 38782*, 59542, 59798, 59799*, 65896*, 65942*, 66452*, 67218*, 67474*, 67735*, 67764*, 67766*, 67991*, 68004*, 68022*, 71297*, 71299*, 71303*, 71342*, 71550*, 71552*, 71553*, 71554*, 71558*, 71566*, 71809*, 71811*, 71824*, 71825*, 71827*, 71845*, 71847*, 71858*, 71860*, 71861*, 71863*, 71913*, 71926*, 72087*, 72101*, 72103*, 72108*, 72160*, 72169*, 72182*, 72356*, 72359*, 72364*, 72380*, 72386*, 72422*, 72630*, 73448*, 75909*, 75931*, 75935*, 75973*, 75975*, 75978*, 75992*, 75995*, 76187*, 76189*, 76191*, 76225*, 76227*, 76232*, 76235*, 76236*, 76251*, 76252*, 76264*, 76270*, 76286*, 76486*, 76732*, 76999*, 77000*, 77003*, 77010*, 77015*, 77016*, 77019*, 77022*, 77030*, 77271*, 77272*, 77273*, 77275*, 77284*
17	36	67176*, 71275*, 71296*, 71304*, 71336*, 71574*, 71810*, 71822*, 71826*, 71831*, 71842*, 71844*, 71910*, 72354*, 73462*, 75911*, 75918*, 75925*, 75927*, 75932*, 75970*, 75972*, 75988*, 76181*, 76224*, 76229*, 76231*, 76244*, 76246*, 76248*, 76738*, 76742*, 76996*, 76998*, 77274*, 381479235*
18	26	67990*, 71273*, 71306*, 71356*, 71815*, 72100*, 72102*, 72406*, 75908*, 75930*, 75974*, 75994*, 75998*, 76183*, 76184*, 76204*, 76206*, 76226*, 76228*, 76238*, 76254*, 76266*, 77002*, 77012*, 77046*, 77270*
19	29	67734*, 71294*, 71298*, 71310*, 71320*, 71322*, 71340*, 71358*, 71838*, 71862*, 72374*, 72438*, 75910*, 75924*, 75934*, 76220*, 76222*, 76230*, 76250*, 77018*, 379234774*, 379251045*, 379311780*, 379513185*, 380492385*, 395241585*, 397207635*, 434234910*, 434365980*
20	26	71272*, 71274*, 71302*, 71846*, 72082*, 72086*, 72116*, 72118*, 72358*, 72390*, 73446*, 76190*, 76502*, 77014*, 379184790*, 379231845*, 379233989*, 379235411*, 379235417*, 379235539*, 379235673*, 379238085*, 379243380*, 379496913*, 397199985*, 434562585*
21	25	71326*, 71582*, 71814*, 72094*, 75990*, 379184670*, 379184677*, 379184745*, 379184775*, 379237575*, 379238853*, 379239525*, 379249815*, 379251030*, 379297380*, 379312740*, 379316580*, 379500225*, 379500900*, 379500993*, 379501665*, 379512465*, 380230215*, 380426775*, 380492433*
22	31	71830*, 72090*, 75926*, 76186*, 76188*, 76234*, 379184662*, 379234140*, 379234241*, 379234257*, 379234260*, 379234759*, 379235537*, 379235779*, 379235793*, 379237740*, 379239495*, 379245101*, 379245221*, 379249005*, 379249731*, 379249763*, 379250229*, 379250281*, 379299796*, 379300293*, 379301313*, 379303617*, 381475365*, 395896350*, 396011220*
23	32	71278*, 76180*, 76182*, 379183691*, 379183751*, 379231585*, 379237830*, 379238599*, 379242932*, 379242933*, 379247013*, 379248956*, 379249181*, 379249261*, 379249687*, 379249725*, 379249747*, 379249799*, 379250325*, 379251033*, 379305153*, 379308468*, 379311012*, 379315764*, 380423025*, 381467715*, 394166551*, 394166760*, 395227860*, 395228370*, 395880135*, 395896343*
24	16	379183725*, 379183973*, 379184700*, 379185507*, 379231593*, 379245165*, 379248925*, 379248985*, 379249213*, 379249301*, 379249437*, 379249500*, 379249557*, 379315860*, 394165992*, 395896373*
25	12	379183916*, 379183917*, 379183980*, 379185453*, 379249497*, 379249556*, 379297128*, 379308964*, 379312548*, 379315092*, 380226405*, 380429145*
26	7	71318*, 379310952*, 379312488*, 379493985*, 380422807*, 380495442*, 396012753*
27	2	380429205*, 395895390*
28	1	395765865*
29	5	379185452*, 379185462*, 380225894*, 380430675*, 395765783*
30	5	379185468*, 380225912*, 381479190*, 395896935*, 396027993*
31	3	394193025*, 396028005*, 417851160*
33	2	379185510*, 380217814*
37	1	379185430*
40	1	380495250*
42	2	380423061*, 380496210*
43	1	1771476585*
46	2	380234070*, 384428310*

References

[Ashenhurst 59] R.L. Ashenhurst. "The Decomposition of Switching Functions," Computation Lab, Harvard University, 29: 74 – 116, 1959.

[Baugh 69] C. R. Baugh, T. Ibaraki, T. K. Liu, and S. Muroga. "Optimum Network Design using NOR and NOR-AND gates by Integer Programming," Dept. Computer Science, University of Illinois, Urbana, Rep. UIUCDCS-R-69-293, Jan. 1969.

[Baugh 71] C. R. Baugh, T. Ibaraki, and S. Muroga. "Results in Using Gomory's All-Integer Integer Algorithm to Design Optimum Logic Networks," *Operations Research*, 19(4): 1090 – 1096, July 1971.

[Baugh 72] C.R. Baugh, C.S. Chandersekaran, R.S. Swee, and S. Muroga, "Optimal Networks of NOR-OR Gates for Functions of Three Variables," *Transaction on Computers*, C-21(2): 153 – 160, February 1972.

[ABC 07] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, Release 70930. http://www.eecs.berkeley.edu/~alanmi/abc/

[Brayton 90] R. Brayton, A. Sangiovanni-Vincentelli, and G. Hachtel, "Multi-level logic synthesis," *Proceeding of the IEEE*, 78(2): 264 – 300, 1990.

[Cormen 01] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms*, Massachusetts: The MIT Press, 2001.

[Coudert 03] O. Coudert, "Two-level Logic Minimization: An overview," *Integration*, 17(2): 97 – 140, 2003.

[Culliney 71] J. N. Culliney. "On the Synthesis by Integer Programming of Optimal NOR Gate Networks for Four Variable Switching Functions," Masters Thesis, Dept. of Computer Science, University of Illinois, Urbana, September 1971.

[Culliney 76] J. N. Culliney, T. Nakagawa, and S. Muroga. "Results of the synthesis of optimal networks of AND and OR gates for four-variable switching functions by a branch-and-bound computer program," Dept. Computer Science, University of Illinois, Urbana, Pre. UIUCDCS-R-76-789, March 1976.

[Culliney 79] J. N. Culliney, M. H. Young, T. Nakagawa, and S. Muroga. "Results of the Synthesis of Optimal Networks of AND and OR Gates for Four-Variable Switching Functions," *IEEE Transactions on Computers*, 27(1): 76 – 85, 1979.

[Curtis 58] H. A. Curtis, "Simple Non-Disjunctive Decomposition," Harvard Computation Laboratory Report No. BLL-19, Sec. II, 1958.

[Curtis 59] H. A. Curtis. "A Functional Canonical Form," *Journal of the ACM*, 6: 245 – 258, 1959.

[Curtis 61] H. A. Curtis. "A Generalized Tree Circuit," *Journal of the ACM*, 8: 484 – 496, 1961.

[Curtis 63] H. A. Curtis. "Generalized Tree Circuit – The Basic Building Block of an Extended Decomposition Theory," *Journal of the ACM*, 10(4): 562 – 581, 1963.

[Davidson 68] E. Davidson, G. Metze. "Module Complexity and NAND Network Design Algorithms," *Annual Allerton Conference on Circuit and System Theory*, 538 – 548, 1968.

[Davidson 68b] E. Davidson. *An Algorithm for NAND Decomposition of Combinational Switching Systems*. PhD Thesis, University of Illinois - Urbana, 1968.

[Davidson 68c] E. Davidson, G. Metze. "Comments on 'An Algorithm for Synthesis of Multiple-Output Combinational Logic,'" *IEEE Transactions on Computers*, 17: 1091 – 1092, 1968.

[Davidson 69] E. Davidson. "An Algorithm for NAND Decomposition Under Network Constraints," *IEEE Transactions on Computers,* C-18(12): 1098 – 1109, 1969.

[Drechsler 98] R. Drechsler, W. Gunther. "Exact Circuit Synthesis," *International Workshop on Logic Synthesis*, 1998.

[Drechsler 98b] R. Drechsler, W. Gunther, "Exact Circuit Synthesis," *Advanced Computer Systems*, 517 – 524, 1998.

[Drechsler 99] R. Drechsler and W. Gunther, "Generation of Optimal Universal Logic Modules," *Proceedings of the 25th EUROMICRO Conference*, 1: 80 – 85, 1999.

[Gunther 99] W. Gunther and R. Drechsler, "Creating Hard Problem Instances in Logic Synthesis Using Exact Minimization," *Proceedings of the IEEE International Symposium on Circuits and Systems*, 6: 436 – 439, 1999.

[Hachtel 96] G.D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*, Massachusetts: Kluwer Academic Publishers, 1996.

[Harrison 63] M.A. Harrison, "The Number of Transitivity Sets of Boolean Functions," *Journal of the Society for Industrial and Applied Mathematics*, 11(3): 806 – 828, 1963.

[Hellerman 63] L. Hellerman. "A Catalog of Three-Variable Or-Invert and And-Invert Logical Circuits," *IEEE Transactions on Electronic Computers*, EC-12(3): 198 – 223, 1963.

[Ibaraki 72] T. Ibaraki, T, K, Liu, C. R. Baugh, and S Muroga, "Implicit enumeration program for zero-one integer programming," *International Journal on Computing Information Science*, 75 – 92, 1972.

[Karp 61] R. M. Karp, F. E. McFarlin, J. P. Roth, and J.R. Wilts. "A Computer Program for the Synthesis of Combinational Switching Circuits," *Proceedings of the AIEE Symposium on Switching Circuit Theory and Logical Design*, 182 – 194, 1961.

[Karp 63] R. M. Karp. "Functional Decomposition and Switching Circuit Design," *Journal of the Society for Industrial and Applied Mathematics*, vol. 11:2, pp. 291 – 335, June 1963.

[Kunz 94] W. Kunz and D.K. Pradhan, "Recursive Learning: A New Implication Technique for Efficient Solutions to CAD-problems: Test, Verification and Optimization," *IEEE Transactio on Conputer-Aided Design*, 13(9): 1143 – 1158, 1994.

[Lai 74] H. C. Lai, T, Nakagawa, and S. Muroga, "Redundancy check technique for designing optimal networks by branch-and-bound method," *International Journal on Computing Information Science* 251 – 271, 1974.

[Lai 79] H.C. Lai and S. Muroga, "Minimum Parallel Binary Adders with NOR (NAND) Gates," *IEE Transactions on Computers*, C-28(9): 648 – 659, 1979.

[Lawler 64] E. L. Lawler. "An Approach to Multilevel Boolean Minimization," *Journal of the ACM*, 11(3): 283 – 295, July 1964.

[McGeer 93] P. McGeer, J. Sanghavi, R. Brayton, and A. Sangiovanni-Vincentelli, "ESPRESSO-SIGNATURE: A new exact minimizer for logic functions," *IEEE Transactions on VLSI Systems*, 1(4): 432 – 440, 1993.

[Muroga 70] S. Muroga. "Logical design of optimal digital networks by integer programming," in *Advances in Information Systems Science*, vol. 3, J. T. Tou, Ed. New York: Plenum, 1970, Chapter 5.

[Muroga 72] S. Murgoa and T. Ibaraki. "Design of Optimal Switching Networks by Integer Programming," *IEEE Transactions on Computers*, C-21: 573 – 582, 1972.

[Muroga 76] S. Muroga and H.C. Lai. "Minimization of Logic Networks Under a Generalized Cost Function," *IEEE Transactions on Computers*, C-25(9): 893 – 907, 1976.

[Muroga 79] S. Muroga. *Logic Design and Switching Theory*, New York: John Wiley & Sons, 1979.

[Nakagawa 71] T. Nakagawa, "A branch-and-bound algorithm for optimal AND-OR networks (The algorithm description)," Dept. Computer Science, Univ. of Illinois, Urbana, Rep. UiUCDCS-R-71-462, June 1971.

[Nakagawa 71b] T. Nakagawa and H, C, Lai, "A branch-and-bound algorithm for optimal NOR networks (The algorithm description)," Dept. Computer Science, Univ. of Illinois, Urbana, rep. UIUCDCS-R-71-438, April 1971.

[Nakagawa 89] T. Nakagawa, H. Lai, S. Muroga. "Design Algorithm of Optimal Logic Networks by the Branch-and-Bound Approach," *International Journal of Computer Aided VLSI Design*, 203 – 231, 1989.

[Niniomiya 61] I. Ninomiya, "A Study of the Structures of Boolean Functions and Its Applications to the Synthesis of Switching Circuits," *Memoirs of the Faculty of Engineering, Nagoya University*, 13(2): 149 – 363, 1961.

[Roth 58] J.P. Roth. "Algebraic Topological Methods for the Synthesis of Switching Systems I," *Transactions of American Mathematical Society*, 88(12), 1958.

[Roth 59] J. P. Roth and E. G. Wagner. "Algebraic Topological Methods for the Synthesis of Switching Systems III. Minimization of Non-singular Boolean Trees," *IBM Journal of Research and Development*, 3(5), 1959.

[Roth 60] J. P. Roth. "Minimization over Boolean Trees," *IBM Journal of Research and Development*, 543 – 558, 1960.

[Roth 62] J. P. Roth and R. M. Karp. "Minimization Over Boolean Graphs," *IBM Journal of Research and Development*, 227 – 238, April 1962.

[Rudell 87] R. Rudell and A. Sangiovanni-Vincentelli, "Multiple-valued minimization for PLA optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-6(5): 727 – 750, 1987.

[Sapra 03] S. Sapra, M. Theobald, and E.M. Clarke, "Sat-based algorithms for logic minimization," *Proceeding of the 21st International Conference on Computer Design*, 510 - 517, 2003.

[Schneider 68] P. R. Schneider and D. L. Dietmeyer. "An Algorithm for Synthesis of Multiple-Output Combinational Logic," *IEEE Transactions on Computers*, C-17(2): 117 – 128, 1968.

[Shannon 49] C. Shannon. "The synthesis of two-terminal switching circuits," *Bell System Technical. Journal*, 28: 59 – 98, 1949.

[Slepian 53] D. Slepian. "On the Number of Symmetry Types of Boolean Functions of *n* Variables," *Canadian Journal of Mathematics*, 5: 185 – 193, 1953.

[Smith 65] R.A. Smith. "Minimal Three-Variable NOR and NAND Logic Circuits," *IEEE Transactions on Electronic Computers*, EC-14(1):79 – 81, 1965.

[Wernick 42] W. Wernick "Complete Sets of Logical Functions," *Transactions of the American Mathematical Society* 51: 117-32, 1942.

[Yang 91] S. Yang, "Logic synthesis and Optimization Benchmarks, Version 3.0," Tech. Report, Microelectronics Center of North Carolina (MCNC), 1991.