Circuit Symmetries in Synthesis and Verification

Donald Chai

Circuit Symmetries in Synthesis and Verification

by

Donald Chai

B.S. (Cornell University) 2001
M.S. (University of California at Berkeley) 2004

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering–Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Fall 2009

The dissertation of Donald Chai is approved:

Chair	Date
	Date
	Date

University of California, Berkeley

Fall 2009

Circuit Symmetries in Synthesis and Verification

Abstract

Circuit Symmetries in Synthesis and Verification

by

Donald Chai
Doctor of Philosophy in Engineering–Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Andreas Kuehlmann, Chair

This dissertation explores the application of logical symmetries in the synthesis and verification of digital systems. Given a high-level description of a design, synthesis algorithms are employed to obtain a low-level description which is suitable for manufacture. To make the process computationally feasible, each step assumes a simplified model of the implementation platform. For example, many of the earlier steps disregard the fact that current technologies require that components are laid out in a two-dimensional plane, and therefore that the necessary wiring between components may become problematic. If the cost and performance requirements are not met, then the synthesis process is repeated, back-propagating the current results to refine any estimates for the next iteration.

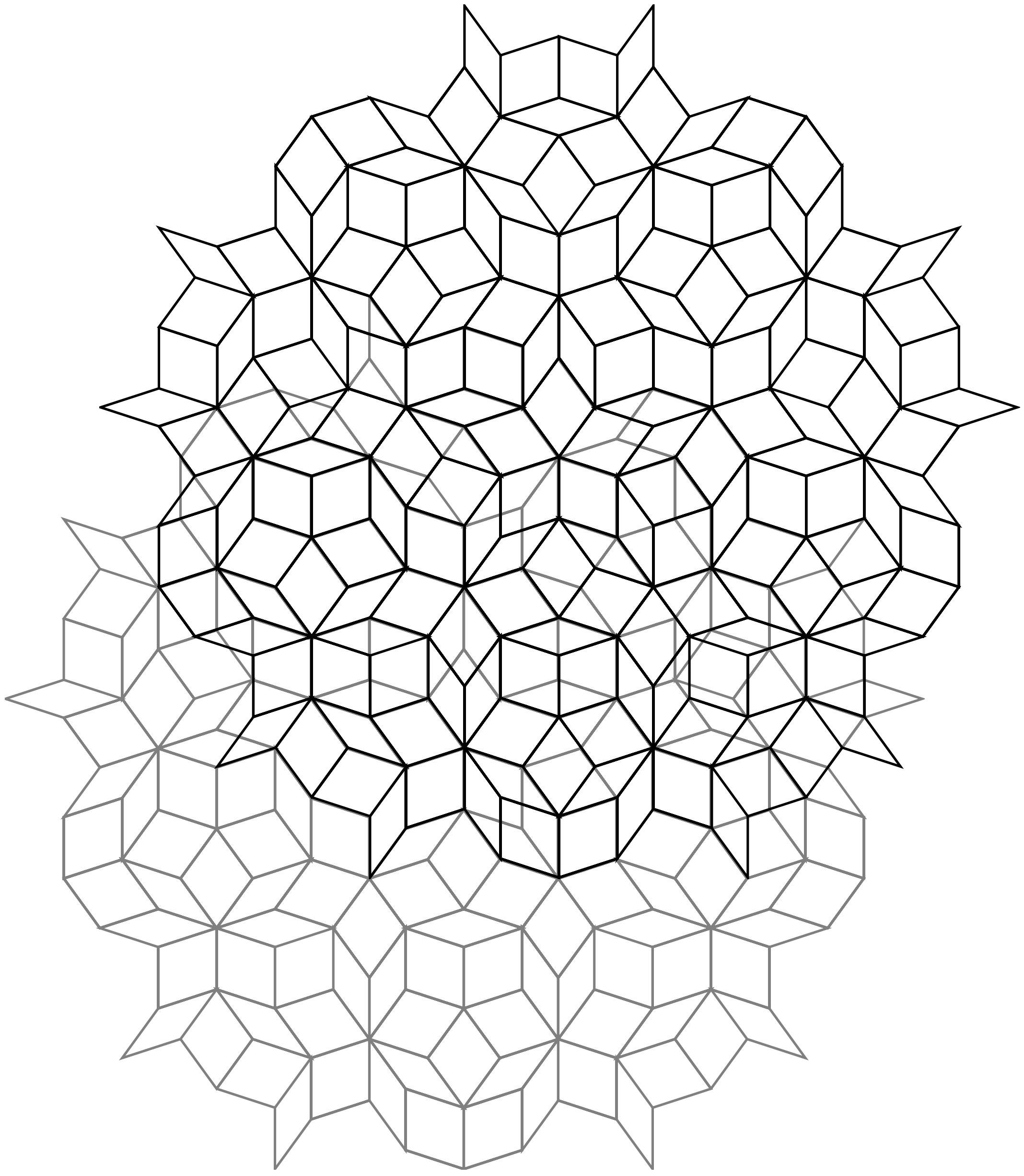
Rather than forcing designers and tools to make early decisions with incomplete and/or inaccurate information, we propose the use of logical symmetries to defer some decisions until more information is available. For example, instead of assuming a fixed circuit structure, we may use symmetries to permute wires during the final stages of synthesis, when wirelengths are known, to improve the design's performance. In addition, symmetries can be used to eliminate redundant cases during verification. For example, in verifying a traffic light controller, one may often assume that the rules are identical for all four directions and simply check for one of them.

The first part of this dissertation reviews the necessary mathematical underpinnings of group theory, allowing us to efficiently reason about symmetries. With this background, we introduce our approaches to find symmetry in circuits.

The second part presents the application of symmetries in synthesis and verification. We show how symmetries may be used in the technology mapping and placement stages, two major steps in synthesis. In technology mapping, we derive alternative representations of the design besides the local minimum obtained from technology independent optimization. In placement, we modify the circuit topology to reduce wirelength. In both cases, symmetries expand the design space with no loss in quality (assuming stable algorithms). Afterwards, we present our approach for improving so-called "symmetry breaking predicates" to speed up Boolean SAT solvers, which are heavily used in verification.

Professor Andreas Kuehlmann
Dissertation Committee Chair

Contents



Two copies of a subregion of the Penrose tiling P3. This tiling is aperiodic, that is, it has no translational symmetry. However any finite subregion appears infinitely often. The overlap between the two copies illustrates this repetition.

Acknowledgments

I would like to thank my advisor Andreas Kuehlmann, who showed me all I know about the research process, from the brainstorming of new ideas to the dissemination of findings to the community at large. A large part of my training has been on the crisp formalization of results. To this day, I do not consider an idea well thought out until it has been well exposited.

I would also like to thank the other members of my dissertation committee, Sanjit A. Seshia and Alper Atamtürk. Professor Seshia is a source of inspiration for me, with his boundless enthusiasm for research. Professor Atamtürk has been a wonderful teacher, and his theoretical and practical approaches to optimization have helped greatly in my research.

In addition, I would like to thank Robert K. Brayton, who had served on my qualification exam committee and co-advised me during my first year at Berkeley. His words have inspired some of the work in Chapter 5.

Besides the professors, one of the advantages to attending a world-class university is in the quality of students that one is able to interact with. I am grateful for the opportunity to meet the students in the DOP center and elsewhere in the department; they have helped me grow both as a researcher and as a human.

The administrative staff at Berkeley have always been a great help in filing paperwork. In particular, I would like to thank Jennifer Stone and Ruth Gjerde for going beyond the normal call of duty and keeping me from panicking over deadlines.

Finally, various other researchers have greatly assisted in performing the research in this dissertation. I would like to thank Alan Mishchenko for his aid in the use of his ABC program, and Kai-Hui Chang and Igor Markov for their help with portions of UMpack (i.e., PhySyn). I would also like to thank Philip Chong for his assistance in running our tool on industrial designs.

1. Introduction

Symmetry, as popularly understood, is the property of self-similarity or invariance under various geometric transformations. A sphere remains the same no matter which way it is rotated. Likewise, a dome remains the same no matter which way it is rotated about the vertical axis. In structural engineering, this property of self-similarity lends strength to a physical system and simplifies design and analysis:

- A basic principle of structural engineering is that all forces and torques in a system must sum to zero to maintain static equilibrium—a nonzero net force implies that the structure is accelerating (i.e. falling). Symmetry guarantees that the top of a dome experience zero net horizontal force.

- Supposing we have a correct design for an arch (a 2-dimensional dome), it is a simple matter to create a dome by superimposing multiple arches which are rotated about the vertical axis.

Exploiting such symmetry is thus one of the reasons why the Roman Pantheon still stands 1900 years after it was built.

Many things in the physical world possess symmetries with respect to one property, but are asymmetric with respect to another. A vase still functions as a vase when rotated about a vertical axis, but one side may be more attractive than another and thus chosen to face the potential viewer. In erecting a building, commodities which are *mostly* interchangeable may be allocated in order to maximize some objective: Miyamoto Musashi's master carpenter assigns material and personnel to best suit the necessary tasks [Miy82]. If Musashi's carpenter faces a new team of workmen, he may first assign them using the information currently available to him, and later reassign the workers as he discovers their specialties. Fortunately for the carpenter, late personnel changes are relatively easy to make.

This dissertation explores the application of logical symmetries in the synthesis and verification of digital systems. As we have just seen, knowledge of symmetries allows us to *enlarge* the design space for various optimization problems (positioning of vase), and to *shrink* the search space for others (analysis of dome).

Outline of this Introduction. To motivate for our work, we first briefly review the steps taken in a traditional electronic design and verification flow which uses computed-aided design (CAD) tools. Then we describe the concept of symmetries as used in this dissertation, and provide examples in which symmetries may be exploited in the CAD flow.

1.1 Contemporary Synthesis and Verification Flow

Given a high-level description of an electronic system, a variety of algorithms are usually employed to arrive at (or "synthesize") a low-level description which is suitable for manufacture on a target platform. Naturally, the final implementation must satisfy certain constraints and optimize certain objectives over cost and performance.

In the case of the Pantheon, Apollodorus of Damascus was given a description, "temple", and synthesized a set of blueprints which could be communicated to craftsmen and which minimized the amount of granite required for construction. The design needed to be verified to ensure that it indeed correctly implemented "temple", and did not instead implement "pile of rubble".

In a digital design scenario, a designer may write a set of Boolean equations describing the behavior of a traffic light controller, and then use synthesis tools to produce a layout of polygons to be drawn on a silicon wafer. Verification tools are used to ensure that the initial set of equations do not allow traffic flows to intersect, and that the final layout is a valid refinement of the initial set of equations.

To make the synthesis task computationally feasible, the process is divided into a series of smaller steps, each of which assumes a simplified model of the final implementation platform. For example, many of the earlier steps disregard the fact that current target platforms require components to be laid out in a two-dimensional plane, and therefore that the necessary wiring between components may become problematic. If the desired cost and performance requirements are not met, then the synthesis process is repeated, backpropagating the current results to refine the estimates for the next iteration. Many iterations may be required to achieve so-called "design closure".

Figure 1.1 shows one possible demarcation of a typical CAD flow for application specific integrated circuits (ASICs), which we will adhere to in the following description. Industrial-scale CAD flows usually consist of many "point tools" which are applied in succession—our description will focus on a few of the major steps. For reference, the introductory chapter of [DGK94] illustrates a similar CAD flow with a different emphasis.

1.1.1 Synthesis

Consider the function f that evaluates to true iff two or three of its four inputs a, b, c, d are true. The designer provides the following register transfer level (RTL) description:

$$f(a,b,c,d) = \bar{a}\bar{b}cd + \bar{a}b\bar{c}d + \bar{a}bc\bar{d} + \bar{a}bcd + a\bar{b}\bar{c}d + a\bar{b}c\bar{d} + a\bar{b}cd + ab\bar{c}\bar{d} + ab\bar{c}d + abc\bar{d}$$

Here $\bar{a}$ denotes the negation of binary variable a (alternatively $\neg a$). The expressions ab and $a + b$ denote the Boolean AND ($a \wedge b$) and OR ($a \vee b$) operations, respectively.

The **technology-independent optimization** stage transforms the above RTL into a **subject graph** consisting of two-input AND and one-input INV (also known as NOT) primitives. As implied by its name, this stage uses a cost model which is independent of target technologies, but shows *some* correlation to the final implementation cost. For our purposes, let the cost of a subject graph be the number of AND primitives in the graph.

Our synthesis tool first simplifies the sum-of-products (SOP) expression we provided by reducing the number of product terms by using an algorithm such as ESPRESSO [BHMSV84]. One possible minimal Boolean expression for f is:

$$f(a,b,c,d) = a\bar{b}c + a\bar{c}d + a\bar{d}b + \bar{a}cd + \bar{a}bd + \bar{a}bc$$

This expression is minimal with respect to the number of sum terms. A different minimal expression might be:

$$f(a,b,c,d) = a\bar{b}d + a\bar{b}c + ab\bar{d} + \bar{a}bc + b\bar{c}d + \bar{b}cd$$

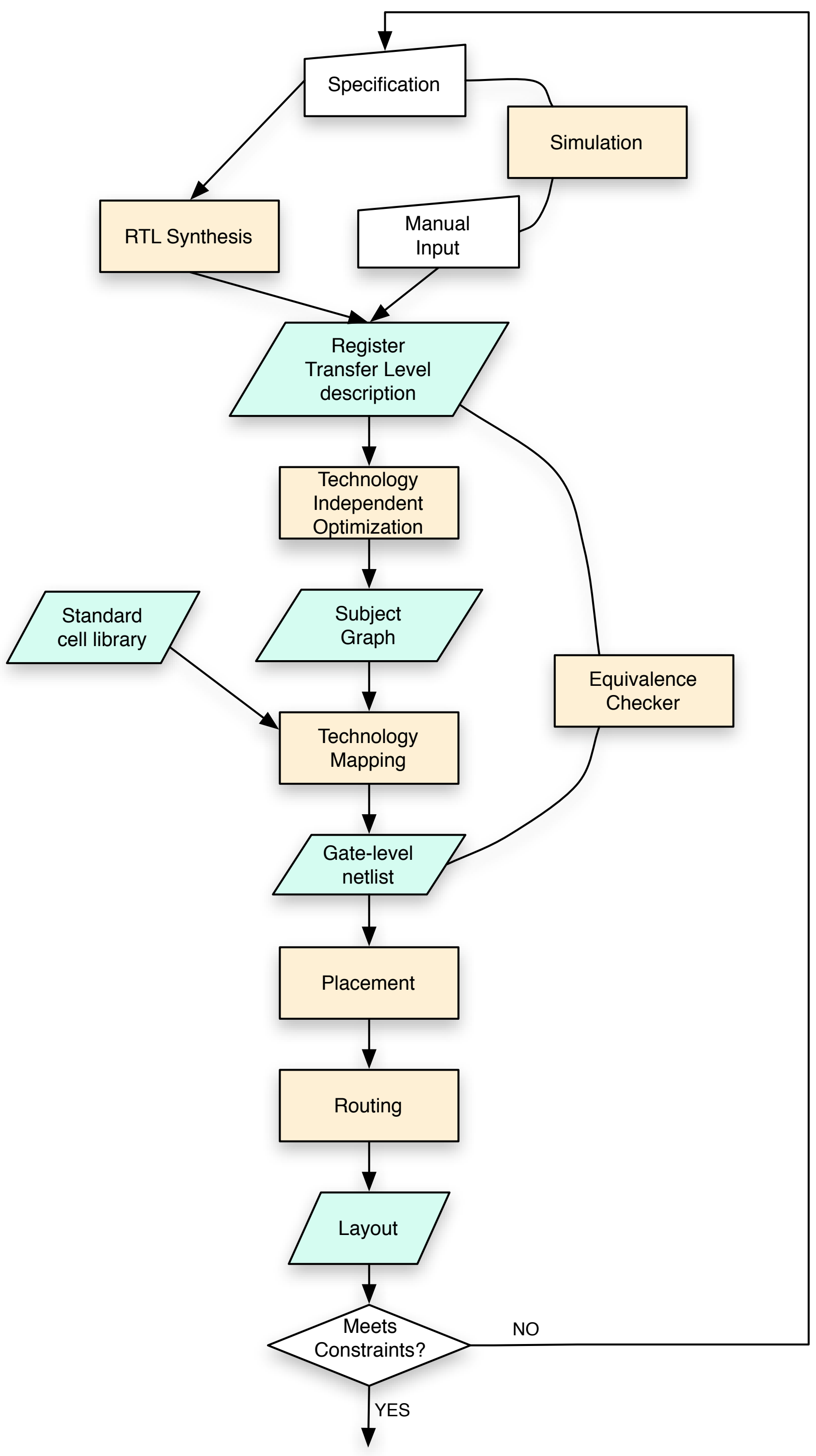


Figure 1.1: Typical CAD flow

The two expressions represent the same function, but are different: the variables appear a different number of times in each. If the values for a and b arrive at different times, one expression may lead to a faster implementation than the other. However, without further information, the synthesis tool can arbitrarily choose either of these as a minimal representation of the function.

Suppose we choose the first SOP. Let us define a **literal** to be a variable or its negation. We extract common factors from the expression to minimize the total number of literals, and reexpress the function as a **Boolean network**, where each **node** implements a local function:

$$\begin{aligned} y_1 &= c + d \\ y_2 &= cd + by_1 \\ y_3 &= \bar{b}c + \bar{c}d + \bar{d}b \\ f &= ay_3 + \bar{a}y_2 \end{aligned}$$

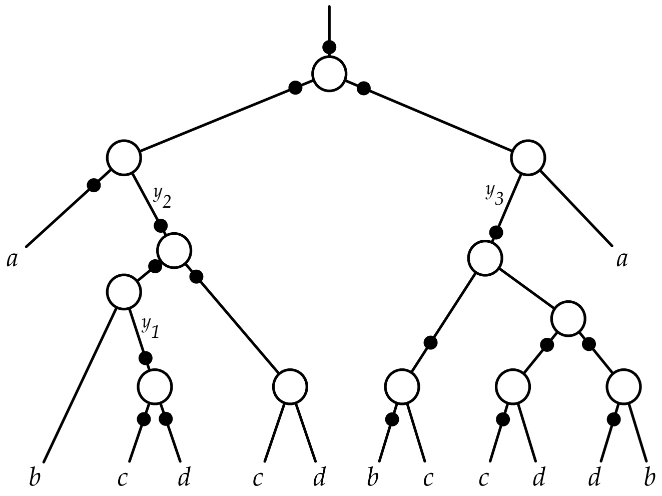
A number of methods may be applied to minimize the network in terms of literal count or the longest path from inputs to output. We ignore them in this discussion because they are not relevant. In our example, the extraction reduced the literal count from 18 to 16.

Given a Boolean network, we convert it into a subject graph (shown in Figure 1.2a) by decomposing each of the constituent nodes into AND and INV primitives. Each product (sum) of more than 3 literals (terms) is arbitrarily decomposed into a tree of AND primitives.

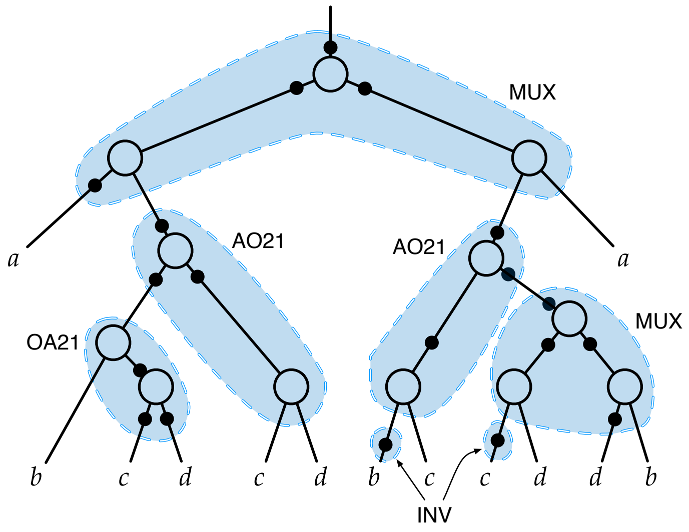
From a subject graph, we find an implementation consisting of standard cells. This is called **technology mapping** (or **library binding**). For example, provided a library containing the following five cells

Cell Name	Function
INV(x_0)	$\bar{x}_0$
AO22(x_0, x_1, x_2, x_3)	$x_0x_1 + x_2x_3$
AO21(x_0, x_1, x_2)	$x_0x_1 + x_2$
OA21(x_0, x_1, x_2)	$(x_0 + x_1)x_2$
MUX(x_0, x_1, x_2)	$x_0x_1 + \bar{x}_0x_2$

we may instantiate cells as in the blue (or shaded) blobs in Figure 1.2b. Each instance of a library cell requires some area, and the technology mapper may aim to find a mapping



(a) Subject graph



(b) Mapping to cell library

Figure 1.2: A subject graph and a potential mapping. Black circles are NOT primitives, larger circles are AND primitives. Pairs of NOTs may be added to allow more cells to match.

which requires the least total cell area. If the only nodes in the subject graph with outdegree greater than 1 are the input nodes, the technology mapping problem for minimal area is solvable in polynomial time [Keu87]. Otherwise, the problem is $\mathcal{NP}$-hard [KR89].

After a Boolean network is mapped into a set of standard cells, the standard cells need to be placed on the plane and wires (interconnect) need to be added to connect the cells. **Placement** and orientation of the cells is a strong factor in determining whether a feasible wiring exists, and if so, the length of the interconnect. Since the delay of a wire is roughly proportional to the square of its length, placement also determines the speed of the final circuit.[1] Thus, placement algorithms typically place the cells so as to minimize the total estimated wirelength. In estimating wirelength, we assume that many wires may pass over the same point, which may not be correct. The actual wirelength is not known until the next stage (**routing**).

At each step of the mapping from Boolean expression to network of standard cells, the synthesis program makes an arbitrary choice, all other things being equal. Rather than forcing designers and tools to make early decisions with incomplete and/or inaccurate information, we propose to use logical symmetries to defer some of the decisions until more information is available. As we will later show, certain symmetries may be analyzed in an efficient manner. Thus, deferring these decisions will not significantly increase the complexity of the synthesis task.

1.1.2 Verification

The results of each step of the synthesis flow must be verified against results from previous steps to guard against errors due to computer bugs or human error. Boolean satisfiability (SAT) forms the cornerstone of many modern verification tools, for example bounded model checkers [BCCZ99] and combinational equivalence checkers [SBSV96]. While the SAT problem is $\mathcal{NP}$-complete, recent advances in SAT solving technology make current solvers scalable for many practical problems [MMZ$^+$01, ES03]

Regardless, current solvers are based on the Davis-Putman-Logemann-Loveland (DPLL) algorithm [DP60, DLL62], and thus require exponential runtime to solve many problems which are otherwise solvable in polynomial time [Hak85, Urq87].

[1]A long wire may be segmented with buffers (also known as repeaters), to improve delay, but this carries with it other costs.

1.2 Symmetry

As we have stated previously, symmetry is the property of invariance under certain transformation. We will also use the term “symmetry” to refer to any such transformation. For a Boolean function, we define symmetry to be invariance of the function under permutation and/or negation of its inputs and/or outputs. This is often referred to as group invariance in the classical literature [McC56], and NPN-equivalence in more recent literature [Mur71, MM90]. For example, the XOR function

$$f = abc\bar{d} + ab\bar{c}d + a\bar{b}cd + a\bar{b}\bar{c}\bar{d} + \bar{a}bcd + \bar{a}b\bar{c}\bar{d} + \bar{a}\bar{b}c\bar{d} + \bar{a}\bar{b}\bar{c}d$$

contains a wealth of symmetries. The inputs may be arbitrarily permuted, and any even number from the set of inputs and output may be negated, e.g. inverting a and the output does not change the function:

$$f = \overline{\bar{a}bc\bar{d} + \bar{a}b\bar{c}d + \bar{a}\bar{b}cd + \bar{a}\bar{b}\bar{c}\bar{d} + abcd + ab\bar{c}\bar{d} + a\bar{b}c\bar{d} + a\bar{b}\bar{c}d}$$

As is the case with other symmetries, **functional symmetries** have the important property that they form a group (in the mathematical sense), that is, any sequence of symmetries may be applied to a function without changing the function. Thus while there are $n!2^n$ valid transformations of the XOR function, we may efficiently manage and manipulate them using results from group theory. For example, we can represent each valid transformation by a composition of elements drawn from a set of n transformations, also known as a generating set.

When reasoning about a logical circuit consisting of multiple levels of logic, the situation is different. Let us refer to the Boolean network for the synthesis example, in which f evaluates to true iff two or three of a, b, c, d are true:

$$\begin{aligned} y_1 &= c + d \\ y_2 &= cd + by_1 \\ y_3 &= \bar{b}c + \bar{c}d + \bar{d}b \\ f &= ay_3 + \bar{a}y_2 \end{aligned}$$

We have already established that the input variables of the function may be arbitrarily permuted without changing the function. In the Boolean network, this might mean, for example replacing every instance of a with b and every instance of b with a as follows:

$$\begin{aligned}
y_1' &= c+d \\
y_2' &= cd + ay_1' \\
y_3' &= \bar{a}c + \bar{c}d + \bar{d}a \\
f' &= by_3' + \bar{b}y_2'
\end{aligned}$$

Since there are intermediate variables in a Boolean network, we can do more than simply modify connections to the primary inputs a, b, c, d for the global function. We may also take any subgraph of the Boolean network, find its functional symmetries, and modify the local connections accordingly. For example, the output of the subgraph encompassing nodes y_1' and y_2' is true iff one or two of a, c, d are true—any of the 6 permutations of a, c, d are correct. The following shows the result after swapping a with c:

$$\begin{aligned}
y_1'' &= a+d \\
y_2'' &= ad + cy_1'' \\
y_3'' &= \bar{a}c + \bar{c}d + \bar{d}a \\
f'' &= by_3'' + \bar{b}y_2''
\end{aligned}$$

The nodes for y_3' and f' are unchanged.

Since many subgraphs may be drawn from a Boolean network, subject graph, or logical circuit, there are ample opportunities to find symmetry—even if a circuit's global function contains none, many of its local functions contain symmetries.

Having established the notion of symmetry in logical circuits, we now provide examples where synthesis and verification may be improved with its use.

1.3 Better CAD through Symmetry

1.3.1 Improving Technology Mapping

The problem with the traditional approach to technology mapping is that the choice of subject graph may cause the resulting mapping to be suboptimal. In the same way that our language shapes our thoughts, the structure of the subject graph has an effect on the final mapping—any instance of a standard cell in the final mapping must correspond to a

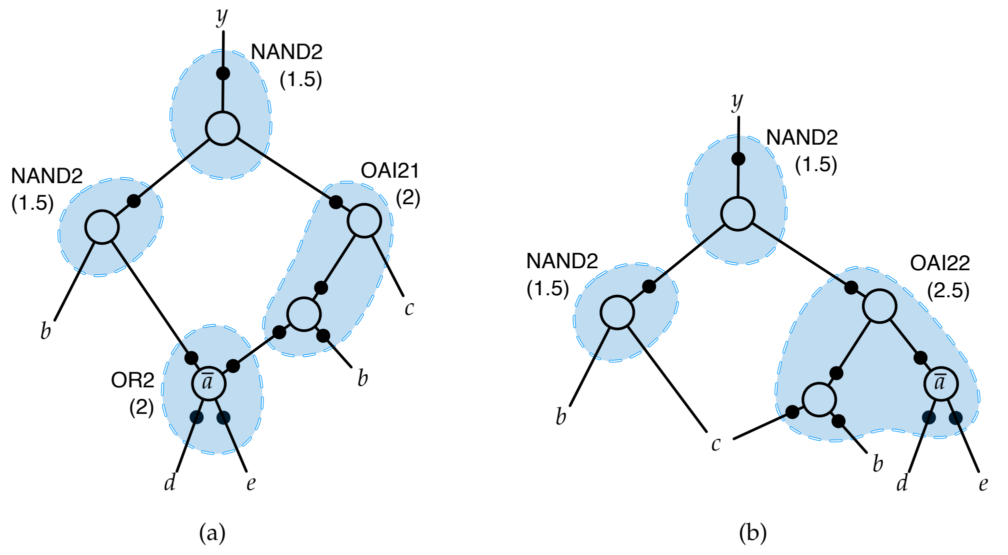


Figure 1.3: Permuting subgraph nodes according to symmetry creates an additional mapping option. Black circles are NOT primitives, larger circles are AND primitives.

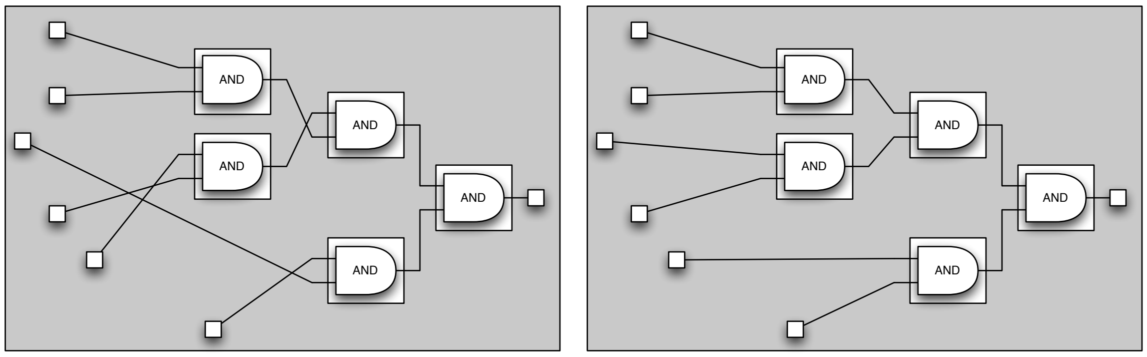
subgraph in the subject graph. Thus, any given subject graph allows only a certain set of mappings.

Figure 5.5a shows a subject graph, where the small black circles are INV primitives, the larger circles are AND primitives, and each shaded region represents a subgraph implemented by a standard cell. By analyzing the symmetries of the subject graph, we find that we can swap *a* and *c*, resulting in Figure 5.5b, which leads to a smaller circuit.

Other modifications to the subject graph are clearly possible, and may result in a smaller circuit as well, but may require repeating some of the technology independent optimization steps. However, iteration is precisely what we are trying to avoid. Chapter 5 describes our approach more fully.

1.3.2 Improving Placement

During the placement stage, we take a network of standard cells and assign the cells to non-overlapping locations in the plane so as to minimize the total estimated wirelength. Figure 1.4a shows a placement of five cells which is optimal assuming that the six inputs and output are fixed, and that the gray region is occupied by other (immobile) cells. The



(a) Placed design before restructuring (b) After restructuring

Figure 1.4: Restructuring a placed design may reduce wirelength.

subcircuit implements a 6-input AND, and we may reassign the connections as long as this functionality is preserved. Figure 1.4b shows the same circuit after having reassigned the connections optimally. Connections to and connections among other cells are not modified, therefore the total estimated wirelength has decreased.

For efficiency, placement algorithms assume that the connectivity of the circuit is fixed. Wirelength may be improved if connections may be modified, either

- after the placement is determined, as in the above example, or
- during the placement step.

Alternatively, a circuit may be modified after placement using **resynthesis** [LESJ98] or **redundancy addition and removal** [JKCMS97]. The difference is that symmetry-based restructuring is more efficient, which allows its use *during* placement. Our restructuring algorithm is described in Chapter 6.

1.3.3 Accelerating SAT

Boolean satisfiability solvers, besides being workhorses in verification, are also increasingly used for various synthesis tasks [NSR99, SVBY06, LJL08] which are in $\mathcal{NP}$ (or co-$\mathcal{NP}$) but for which no polynomial time algorithm is known. Thus any improvements in SAT solving technology would have a great effect on many different stages of the CAD flow.

The DPLL algorithm [DP60, DLL62] used in all complete SAT solvers combines a branch-and-bound search with the resolution rule of logic. At each step of the search, a SAT solver assigns a value to some variable which has not already been assigned one. If any of the constraints are not satisfied, modern solvers record a nogood [SS77] using the resolution rule [MSS99] and subsequently backtrack.

Suppose we are given an instance of the pigeonhole problem with 11 pigeons and 10 holes, and are told to assign each pigeon to a hole without sharing. After we encode the instance as SAT and input the instance to a SAT solver, the solver:

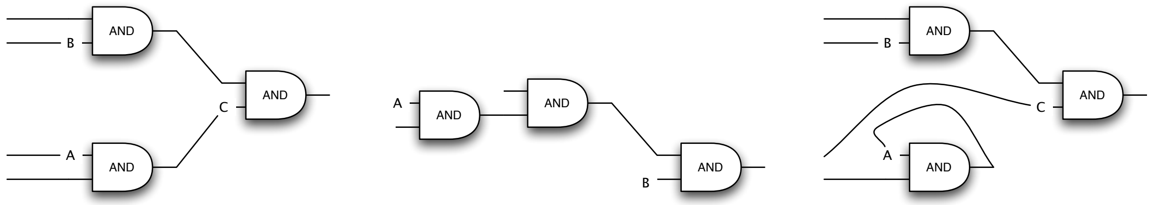
- puts pigeon 1 in hole 1 and fails to assign the remaining 10 pigeons to the remaining 9 holes,
- puts pigeon 1 in hole 2 and fails to assign the remaining 10 pigeons to the remaining 9 holes,
- puts pigeon 1 in hole 3 and fails to assign the remaining 10 pigeons to the remaining 9 holes,
- . . .
- puts pigeon 1 in hole 10 and failing to assign the remaining 10 pigeons to the remaining 9 holes, finally quits.

Since the pigeons and holes are identical, we can safely conclude that the problem is unsatisfiable after putting pigeon 1 in hole 1, and not try putting pigeon 1 in any other hole. Then, we assume without loss of generality that pigeon 2 is in hole 2, and so on. Using this reasoning, we can prove unsatisfiability in linear time rather than in exponential time [Hak85].

The general approach which we will follow is that of [CGLR96], which adds **symmetry breaking predicates** to a SAT formula in order to eliminate redundant parts of the search space. Chapter 7 describes our approach to formulate symmetry breaking predicates.

1.4 Challenges to Solve

Since our proposed approach of using symmetries is one of many competing solutions for solving the synthesis and verification problems, we must tackle the perennial tradeoffs



(a) (b) Swap B with C (c) Swap A with C

Figure 1.5: Symmetries within a circuit do not form a group.

between quality of results and runtime.

- The problem of finding symmetries in Boolean functions has not been completely solved. Current solutions either specialize for the case of functions with one output [AP05, CK06, ABPS07, KK08] or reduce the problem to an instance of graph isomorphism using an exponentially-sized graph [CMB05a].

- Symmetries within a circuit do not form a group. In the circuit in Figure 1.5a, connection A may be swapped with connection B, and connection B may be swapped with connection C Figure 1.5b. However, swapping A with C is incorrect, because it creates a cycle (Figure 1.5c). Thus, algorithms to reason about functions cannot be used without modification for circuits.

- A large number of subgraphs can be extracted from any Boolean network, subject graph, or logical circuit for analysis of symmetries. Finding all symmetries may not be practical, and therefore we must isolate those which provide the most potential for optimization.

- After finding symmetries, we must be able to efficiently apply them in synthesis. Explicit enumeration of permutations is not practical.

- The use of symmetries in SAT is purely to reduce the amount of time needed to solve a given SAT formula. It is possible that finding symmetries or generating effective symmetry breaking predicates may take longer than solving the SAT problem directly to begin with. Another possibility is that the augmented SAT problem may be more difficult to solve due to the sheer number of added predicates.

1.5 Contributions of this Dissertation

This dissertation presents the following contributions toward the utilization of symmetries for synthesis and verification:

- An efficient method to find symmetries in functions which have multiple outputs. Our method is based on a reduction to *small* instances of graph isomorphism which can be solved for practical functions.
- An approach to finding symmetries in a circuit by analyzing the circuit's structure to locate subgraphs which are likely to contain symmetries that are productive for optimization.
- A general approach for applying symmetries in technology mapping and placement by decomposing a group into orthogonal subgroups, each of which may be explored efficiently using a specialized procedure. Enumeration is thus limited to the number of subgroups, rather than to the total number of symmetries.
- An efficient preprocessor for generating symmetry breaking predicates for SAT formulas. Our preprocessor analyzes the structure of a formula's symmetry group in order to produce a small set of predicates that breaks as many symmetries as practically possible.

1.6 Organization of this Dissertation

This dissertation is organized into two parts:

- Part I provides the framework for finding symmetries in circuits. Chapter 2 reviews basic group theory and algorithms for permutation groups. Chapter 3 describes a procedure for finding symmetries of multiple-output functions. Chapter 4 describes our subgraph selection heuristic for finding symmetries in circuits.
- Part II applies the background established in Part I towards three problems in CAD. Chapters 5, 6, and 7 discuss technology mapping, placement, and SAT, respectively.

Finally, the conclusion summarizes our contributions and points out their strengths and weaknesses.

2. Permutation Group Theory

In this dissertation, the concept of symmetry refers to the ability to permute connections in a circuit without affecting its logical behavior. Permutation group theory is a mature branch of mathematics, and any efficient algorithms which use symmetries will rely on it. Therefore, this chapter serves to provide a sufficient background for the remainder of this dissertation. The reader is referred to [Hal59, Ser02] for a more thorough treatment.

2.1 Basic Definitions

2.1.1 Groups

A **group** is a set G that together with a binary operation $\cdot : G \times G \to G$, satisfies four group axioms:

1. Closure: for any a, b in G, $a \cdot b \in G$.
2. Associativity: for any $a, b, c \in G$, $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
3. Identity: there exists some $i \in G$ such that for any $x \in G$, $i \cdot x = x \cdot i = x$.
4. Inverse: for any element $a \in G$, there exists an element $b \in G$ such that $a \cdot b = b \cdot a = i$.

Example 2.1. Let us define a binary operation $\cdot$ over the set $Z_6 = \{1, 2, 4, 8, 16, 32\}$, such that $a \cdot b = 2^{\lg(ab) \mod 6}$, where lg is the base-2 logarithm. The following "multiplication table" (or **Cayley table**) shows the operation explicitly:

$\cdot$	1	2	4	8	16	32
1	1	2	4	8	16	32
2	2	4	8	16	32	1
4	4	8	16	32	1	2
8	8	16	32	1	2	4
16	16	32	1	2	4	8
32	32	1	2	4	8	16

The set Z_6 forms a group with $\cdot$ since it satisfies the four group axioms:

1. Closure: from the definition of $\cdot$, if a and b are powers of 2, then $a \cdot b$ is also a power of 2 between 2^0 and 2^5, inclusive.

2. Associativity: from the associativity of addition

$$2^{\lg(ab)+\lg(c) \mod 6} = 2^{\lg(a)+\lg(bc) \mod 6}$$

3. Identity: $x \cdot 1 = x$ for any $x \in Z_6$.

4. Inverse: $2 \cdot 32 = 4 \cdot 16 = 8 \cdot 8 = 1$.

We use multiplicative conventions to refer to the binary operation $\cdot$, i.e. $a \cdot b$ (or simply ab) is the **product** of a and b. Its exact definition will be understood from its context. Since $\cdot$ is associative, we omit parentheses from algebraic expressions. The group axioms imply that each element has a unique inverse, and the inverse of a is denoted by a^{-1}.

As a consequence of the group axioms, a group may be described implicitly by a small set of **generators**. A set K generates group G, i.e. $\langle K \rangle = G$, if every $a \in G$ can be written as a product of elements from K, and if every product of elements in K is also in G.

Example 2.2. The group Z_6 from the previous example is generated by the set $\{2\}$:

$$\begin{aligned} 2 &= 2 \\ 4 &= 2 \cdot 2 \\ 8 &= 2 \cdot 2 \cdot 2 \\ 16 &= 2 \cdot 2 \cdot 2 \cdot 2 \\ 32 &= 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \\ 1 &= 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \end{aligned}$$

A **subgroup** H of a group G, is a subset of G that with the same operation, satisfies the four group axioms. This relationship is denoted by $H \leq G$. For example, the set $\{1,8\}$ forms a subgroup of Z_6.

For groups G, H, where $H \leq G$, the **(right) coset** of H for some $x \in G$ is defined as $\{hx : h \in H\}$ and denoted by Hx. The cosets of H form equivalence classes. A set consisting of an element from each class is called a **(right) transversal** of H, denoted by $G : H$. The size of the transversal is denoted by $|G : H|$, and Lagrange's Theorem states that $|G| = |G : H||H|$. In other words, the cosets form a partition of G.

Example 2.3. Suppose that $G = Z_6$ and $H = \{1,8\}$. We enumerate the cosets Hx for each $x \in G$ as follows:

Cosets Hx		
$\{1,8\}\cdot 1$	$=$	$\{1 \cdot 1, 8 \cdot 1\} = \{1,8\} = H$
$\{1,8\}\cdot 2$	$=$	$\{1 \cdot 2, 8 \cdot 2\} = \{2,16\}$
$\{1,8\}\cdot 4$	$=$	$\{1 \cdot 4, 8 \cdot 4\} = \{4,32\}$
$\{1,8\}\cdot 8$	$=$	$\{8,1\}$
$\{1,8\}\cdot 16$	$=$	$\{16,2\}$
$\{1,8\}\cdot 32$	$=$	$\{32,4\}$

The table shows 3 distinct cosets which each appear 2 times. This conforms to Lagrange's Theorem—since $|G| = 6$ and $|H| = 2$, $|G : H| = 3$. Here, the cosets of H are H, $\{2,16\}$, and $\{4,32\}$; the set $\{1,2,4\}$ forms a transversal. Another way to show this is to represent each element $g \in G$ as a product hx of some $h \in H$ and $x \in \{1,2,4\} = (G:H)$.

g	$=$	$h \cdot x$
1	$=$	$1 \cdot 1$
2	$=$	$1 \cdot 2$
4	$=$	$1 \cdot 4$
8	$=$	$8 \cdot 1$
16	$=$	$8 \cdot 2$
32	$=$	$8 \cdot 4$

Left cosets and left transversals are defined similarly, e.g. $xH = \{xh : h \in H\}$, with identical properties. We will not be using the "left" variants, but introduce them in order to caution against their inadvertent use. Since a left (right) transversal may be obtained from a right (left) transversal by inverting each element, it is easy to confuse the two. (The difference will be apparent later, when the binary operation is not commutative.)

As we will see, symmetries form a group. These results imply that symmetries may be represented compactly, and that symmetries may be partitioned into a disjoint set of cosets for efficient processing.

2.1.2 Permutations

Let $\Omega = \{1,2,\ldots,n\}$ be a finite set of **points**. A **permutation** π is a bijection from Ω to itself. Permutations may be written in cyclic notation, e.g. the permutation $(1,3)(2,4,5)$ swaps 1 with 3 and simultaneously **moves** 2 to 4, 4 to 5, and 5 to 2. The identity permutation is denoted by $()$. The **image** of a point i under π is written i^π, e.g. $5^{(1,3)(2,4,5)} = 2$. The set of **moved points** of a permutation is the set $\{i \in \Omega : i^\pi \neq i\}$; conversely, a permutation **(pointwise) stabilizes** a set if none of the points in the set are moved. For example, the permutation $(1,3)(2,4,5)$ moves $\{1,2,3,4,5\}$ and stabilizes $\{6,7,\ldots,n\}$.

Permutations may also be written in Cartesian notation, which lists the images of each point. For example, $(1,3)(2,4,5)$ may be written as (34152), and $()$ may be written as (12345). Unspecified images are denoted by dashes, e.g. if $\pi = (53\text{ - - -})$ then $1^\pi = 5$, $2^\pi = 3$, and $3^\pi, 4^\pi, 5^\pi$ are unspecified. We will use this notation when permutations are partially specified or when enumerating elements of a permutation group. Otherwise, we prefer to use cyclic notation because properties such as the set of moved points can be determined by inspection.

2.1.3 Permutation Groups

A **permutation group** is a group consisting of permutations with a binary operation defined as follows. Given permutations π_1 and π_2, their product $\pi_1\pi_2$ is defined such that if $i^{\pi_1} = j$ and $j^{\pi_2} = k$, then $i^{(\pi_1\pi_2)} = k$. Since $i^{(\pi_1\pi_2)} = (i^{\pi_1})^{\pi_2}$ for any point i, we may omit the parentheses and simply write $i^{\pi_1\pi_2}$ with no ambiguity.

Example 2.4. Suppose that $\pi_1 = (1,3)(2,4,5)$ and $\pi_2 = (1,3)(2,5,4)$, then $\pi_1\pi_2 = ()$, since $1^{\pi_1} = 3$ and $3^{\pi_2} = 1$, $2^{\pi_1} = 4$ and $4^{\pi_2} = 2$, etc.

The group consisting of all $n!$ permutations over n points is denoted by S_n, and called the **symmetric group** when n or Ω are known from the context.

A permutation group G induces a partition of Ω, known as the set of **orbits**; for $x \in \Omega$, $\text{Orbit}(x) = \{x^\pi : \pi \in G\}$.

Example 2.5. Let $n = 11$ and $G = \langle\{(1,3)(2,4,5),(6,7)(9,11),(7,8)\}\rangle$. The orbits of G are $\{\{1,3\},\{2,4,5\},\{6,7,8\},\{9,11\},\{10\}\}$, and can be obtained using a union-find algorithm [CLR89] on the cycles of the generators—the generator $(6,7)(9,11)$ specifies that 6 may move to 7, and $(7,8)$ specifies that 7 may move to 9. By transitivity, 6 may move to 9, therefore 6 and 9 are in the same orbit.

Note that the product operation over permutations is *not* commutative, and the following example illustrates the difference between left and right cosets.

Example 2.6. Suppose $G = S_3$ and $H = \{(),(2,3)\}$. A right transversal is $\{(),(1,2),(1,3,2)\}$, and a left transversal is $\{()^{-1},(1,2)^{-1},(1,3,2)^{-1}\}$. The following list shows how G may be partitioned into left cosets and right cosets:

Left cosets xH		G		Right cosets Hx
$()\cdot()$	$=$	(123)	$=$	$()\cdot()$
$()\cdot(2,3)$	$=$	(132)	$=$	$(2,3)\cdot()$
$(1,2)\cdot()$	$=$	(213)	$=$	$()\cdot(1,2)$
$(1,2,3)\cdot()$	$=$	(231)	$=$	$(2,3)\cdot(1,2)$
$(1,2)\cdot(2,3)$	$=$	(312)	$=$	$()\cdot(1,3,2)$
$(1,2,3)\cdot(2,3)$	$=$	(321)	$=$	$(2,3)\cdot(1,3,2)$

Note that two elements from a right transversal, $\{(1,2),(1,3,2)\}$ lie in the same left coset $(1,2)H$.

2.2 Stabilizer Chains

Given a permutation group G, we would like to formulate efficient algorithms to:

- compute $|G|$
- enumerate the elements of G
- compute a small set of generators
- check whether a permutation π is in G
- compute the transversal of any given subgroup

Most efficient algorithms [Ser02] rely on the concept of a stabilizer chain for G. This is a successively shrinking series of subgroups which stabilizes a growing corresponding sequence of points. A formal definition is given below.

For a group G and set of points S, the stabilizer G_S of S in G is the largest subgroup of G that pointwise stabilizes S, that is $G_S = \{g : g \in G \wedge \bigwedge_{s \in S} s^g = s\}$. A **base** B is a sequence of points $\beta_1, \beta_2, \ldots, \beta_k$ such that G_B contains only the identity permutation. Let $G^{(i)}$ denote $G_{\{\beta_1, \beta_2, \ldots, \beta_i\}}$ and let $G^{(0)} = G$. B induces a **stabilizer chain** as follows:

$$\{()\} = G^{(k)} \leq G^{(k-1)} \leq \cdots \leq G^{(2)} \leq G^{(1)} \leq G^{(0)} = G$$

The base is **reduced** if $G^{(i)} \neq G^{(j)}$ for $i \neq j$. Given a group G and base B, a **strong generating set** $T \subseteq G$ is one for which

$$\langle T \cap G^{(i)} \rangle = G^{(i)}$$

for all i.

Example 2.7. Suppose $G = S_5$. The sequence $1,2,3$ is *not* a base for G because two permutations in G stabilize $\{1,2,3\}$: $()$ and $(4,5)$.

Example 2.8. Suppose $G = S_5$ and $B = 1, 2, 3, 4$. The stabilizer chain is as follows:

$$\begin{aligned}
G^{(4)} &= \{()\} \\
G^{(3)} &= \{(), (4,5)\} \\
G^{(2)} &= \text{all 6 permutations over } \{3,4,5\} \\
G^{(1)} &= \text{all 24 permutations over } \{2,3,4,5\} \\
G^{(0)} &= \text{all 120 permutations over } \{1,2,3,4,5\}, \text{i.e. } S_5
\end{aligned}$$

The permutations $\{(1,2), (1,2,3,4,5)\}$ generate S_5, but are not a strong generating set for base B since $G^{(3)} = \{(), (4,5)\}$ but $\langle \{(1,2), (1,2,3,4,5)\} \cap \{(), (4,5)\}\rangle = \langle\{()\}\rangle = \{()\} \neq G^{(3)}$. T_0, defined below, is a strong generating set for S_n

$$\begin{aligned}
T_4 &= \{()\} \\
T_3 &= T_4 \cup \{(4,5)\} \\
T_2 &= T_3 \cup \{(3,4,5)\} \\
T_1 &= T_2 \cup \{(2,3)\} \\
T_0 &= T_1 \cup \{(1,2)\}
\end{aligned}$$

since $\langle T_i \rangle = G^{(i)}$.

In the remaining discussion, the base will be assumed to be the sequence of points $1, 2, \ldots, n$ for simplicity and for consistency with [Jer86].

Sims [Sim71] presented the first algorithm for finding a strong generating set based on a theorem by Schreier [Hal59]. Efficient variations of the so-called Schreier-Sims method are described in [Jer86] and [Knu91], which present worst-case bounds of $O(n^5)$ runtime. Asymptotic complexity can be improved by adding Monte Carlo methods [BCF$^+$91], at the expense of possibly computing generators for a subgroup of G, rather than for the entire group.

Before presenting algorithms to compute $|G|$ or check for membership, we describe the main results of the algorithm proposed by [Jer86]. Compared to other variants of the Schreier-Sims method, this algorithm is more comprehensible for readers who are already familiar with graph theory.

2.3 Jerrum's Branching Structure

We focus on the general approach described in [Jer86], which produces a small generating set ($|T| < n$) in the form of a "labeled branching". The labeled branching decomposes G into transversals: $G = (G^{(n-2)} : G^{(n-1)}) \cdots (G^{(2)} : G^{(3)})(G^{(1)} : G^{(2)})(G^{(0)} : G^{(1)})$. Since $G^{(i)} = G^{(i-1)}(G^{(i)} : G^{(i-1)})$ and $G^{(n-1)} = ()$, the transversals form a generating set for each $G^{(i)}$, i.e., a strong generating set for G. We now show how Jerrum's branching represents these transversals.

The labeled branching (actually a forest) maintains certain structural properties:

1. Nodes are synonymous with points in Ω, and are ordered $1 < 2 < \cdots < n$.
2. Edges are directed and follow the node ordering: an edge connecting nodes i, j where $i < j$ is necessarily from i and to j.
3. An edge from i to j is labeled with a permutation $\sigma_{i,j} \in T$ where $i^{\sigma_{i,j}} = j$ and all points less than i are stabilized ($a^{\sigma_{i,j}} = a$ for all $a < i$).

Thus a path from i to m ($i \to j \to k \to \cdots \to l \to m$) can be construed as a permutation π where $\pi = \sigma_{i,j}\sigma_{j,k} \cdots \sigma_{l,m}$, $i^{\pi} = m$, and all points less than i are stabilized. A path from i to m exists if and only if $m \in \{i^{\pi} : \pi \in G^{(i-1)}\}$. In other words, the set of points reachable from point i (including i itself) is equal to the orbit of i in subgroup $G^{(i-1)}$, and the corresponding paths form the transversal $G^{(i-1)} : G^{(i)}$.

As we stated previously, any permutation $\pi \in G^{(i)}$ can be represented by some product of elements from the transversals:

$$G^{(i)} = (G^{(n-2)} : G^{(n-1)}) \cdots (G^{(i)} : G^{(i+1)})(G^{(i-1)} : G^{(i)})$$

Therefore, the edge labels of the subgraph induced by nodes $\{i, i+1, \ldots, n\}$ generate $G^{(i-1)}$, and the edge labels of the entire branching form a strong generating set for G.

Example 2.9. Let $G = \langle\{(1,2),(1,3)(2,4)\}\rangle$. This group represents the valid permutations of the variables in the algebraic expression $x_1x_2 + x_3x_4$. Assuming that variable x_i is represented by point i, $x_2x_1 + x_3x_4$ is an equivalent expression represented by permutation $(1,2)$. The expression $x_3x_4 + x_1x_2$ is also equivalent, represented by $(1,3)(2,4)$. Finally, $x_1x_2 + x_4x_3$ is also equivalent, represented by $(3,4)$.

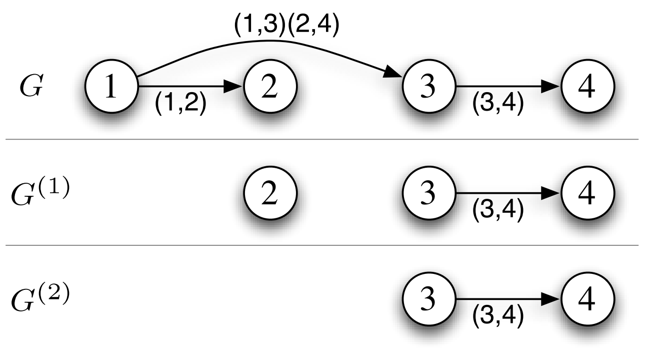


Figure 2.1: Labeled branchings for $G = \langle\{(1,2),(1,3)(2,4)\}\rangle$ and subgroups $G^{(1)}$ and $G^{(2)}$

The stabilizer chain for G is as follows:

$$\begin{aligned}
G^{(3)} &= \{()\} \\
G^{(2)} &= \{(),(3,4)\} \\
G^{(1)} &= \{(),(3,4)\} \\
G = G^{(0)} &= \{(),(1,2),(3,4),(1,2)(3,4),(1,3)(2,4),(1,3,2,4),(1,4,2,3),(1,4)(2,3)\}
\end{aligned}$$

Figure 2.1 shows the labeled branching for G, and the induced subgraphs for $G^{(1)}$ and $G^{(2)}$. From Lagrange's theorem, we know that $|G^{(0)} : G^{(1)}| = \frac{|G^{(0)}|}{|G^{(1)}|} = 4$, but do not know the elements of $G^{(0)} : G^{(1)}$. The paths from each point i form $G^{(i-1)} : G^{(i)}$ as follows:

$$\begin{aligned}
G^{(2)} : G^{(3)} &= \{(),\ \sigma_{3,4}\} = \{(),(3,4)\} \\
G^{(1)} : G^{(2)} &= \{()\} \text{ since } G^{(1)} = G^{(2)} \\
G^{(0)} : G^{(1)} &= \{(),\ \sigma_{1,2},\ \sigma_{1,3},\ \sigma_{1,3}\sigma_{3,4}\} = \{(),(1,2),(1,3)(2,4),(1,4,2,3)\}
\end{aligned}$$

Note that each node in a labeled branching has indegree at most 1. For efficiency, node j is labeled with a permutation τ_j where $\tau_j = \tau_i\sigma_{i,j}$ if an edge $\sigma_{i,j}$ exists, otherwise $\tau_j = ()$. Thus for any path $i \rightsquigarrow m$, the product $\sigma_{i,j}\sigma_{j,k}\cdots\sigma_{l,m}$ can be computed in terms of its endpoints: $\tau_i^{-1}\tau_m$.

Our discussion centers on how to *use* a labeled branching, and we refer the reader to [Jer86] for details on how to *construct* a labeled branching.

Going further, we can calculate $|G|$ recursively as $|G^{(i)}| = |G^{(i)} : G^{(i+1)}||G^{(i+1)}|$. We can also randomly select a permutation from G by choosing a random element from

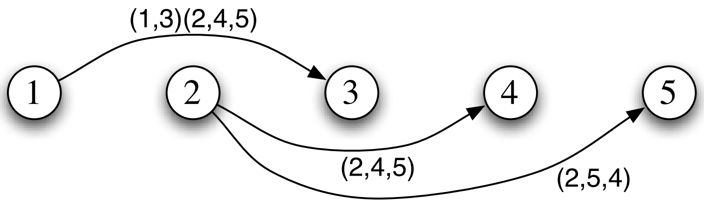


Figure 2.2: Labeled branching for group $\langle\{(1,3)(2,4,5)\}\rangle$

each $G^{(i)} : G^{(i+1)}$ for $i = 0, 1, \ldots, n-1$ and taking their product.

Example 2.10. Let $G = \langle\{(1,3)(2,4,5)\}\rangle$, and let the graph in Figure 2.2 be a labeled branching for G. Point 1 can reach points 1 and 3, and point 2 can reach points 2, 4, and 5, indicating that $|G| = 2 \cdot 3 = 6$. We can also enumerate the 6 elements of G as products of elements from each transversal:

$$\begin{aligned}
() \cdot () &= () \\
(2,4,5) \cdot () &= (2,4,5) \\
(2,5,4) \cdot () &= (2,5,4) \\
() \cdot (1,3)(2,4,5) &= (1,3)(2,4,5) \\
(2,4,5) \cdot (1,3)(2,4,5) &= (1,3)(2,5,4) \\
(2,5,4) \cdot (1,3)(2,4,5) &= (1,3)
\end{aligned}$$

Note that $(1,3)$ is another valid label for the edge $\sigma_{1,3}$. This will be an important point in a chapter 7.

2.3.1 Membership Check

Checking for membership utilizes a "sifting" procedure which resembles algebraic division, or more accurately, the solution of a Rubik's cube. The solution of a Rubik's cube entails restoring one layer at a time until the identity permutation is reached. Similarly, the sifting procedure systematically attempts to "undo" the movement of points 1, 2, 3, etc.

Example 2.11. Using the same group and labeled branching from the previous example, let us check whether $(1,3) \in G$. Since $(1,3)$ is not the identity, we multiply by $\tau_3^{-1}\tau_1$, resulting in $(2,5,4)$. The result is not the identity either, and we multiply it by $\tau_5^{-1}\tau_2$, resulting in $()$. Finally, we conclude that $(1,3) \in G$.

Algorithm 1 IS-MEMBER(G, π)

```
1: while $\pi \neq ()$ do
2:     $j \leftarrow$ the first moved point of $\pi$
3:     $k \leftarrow j^{\pi}$
4:     if there is no path from $j$ to $k$ then
5:         return false
6:     end if
7:     $\pi \leftarrow \pi\tau_k^{-1}\tau_j$ { undo movement from $j$ to $k$: $(\tau_j^{-1}\tau_k)^{-1} = \tau_k^{-1}\tau_j$ }
8: end while
9: return true
```

Example 2.12. Let us now check whether $(2,4) \in G$. The first moved point is 2 and $2^{(2,4)} = 4$, so we combine with $\tau_4^{-1}\tau_2$, resulting in $(4,5)$. Since there is no path in the graph from 4 to 5, we conclude that $(2,4) \notin G$.

2.3.2 Transversal Computation

Jerrum described in [Jer86] how to obtain a transversal $S_n : G$ given a branching structure for G—the set of topological sorts of the branching structure forms a transversal. We can extend this concept for any arbitrary G and H where $H \leq G$—the set of elements in G consistent with a topological ordering of H's branching structure forms a transversal $G : H$. Note that the labels for the subgroup's branching structure are not required.

Many group algorithms follow a similar recursive structure. Therefore, for illustrative purposes, we first present a naïve version of the algorithm and add optimizations incrementally. For brevity, the pseudocode equates a group with its respective branching structure.

Algorithm 2 TRANSVERSAL(G, H)

```
1: $L \leftarrow \{\}$
2: TRANSVERSAL-RECUR($H$, 1, ())
3: return $L$
```

The naïve version enumerates the elements of G, and selects those elements which obey the topological ordering imposed by H's branching structure.

Suppose that $G = \langle\{(1,3),(2,4),(4,5)\}\rangle$ and $H = \langle\{(2,4,5)\}\rangle$. Their corresponding branching structures are shown in Figure 2.3, and indicate that $|G| = 12$ and $|H| = 3$. Therefore, we expect that the algorithm produce a set of $12/3 = 4$ permutations.

Algorithm 3 TRANSVERSAL-RECUR(G, H, j, π)

```
if $j = n$ then
    if $k^\pi < l^\pi$ for every edge $(k, l)$ in $H$ then
        { $\pi$ is consistent with a topological sort of $H$ }
        $L \leftarrow L \cup \{\pi\}$
    end if
else
    $\tau \leftarrow$ node labels from $G$
    for each $k$ reachable from $j$ in $G$ do
        TRANSVERSAL-RECUR($G$, $H$, $j + 1$, $\tau_j^{-1}\tau_k\pi$)
    end for
end if
```

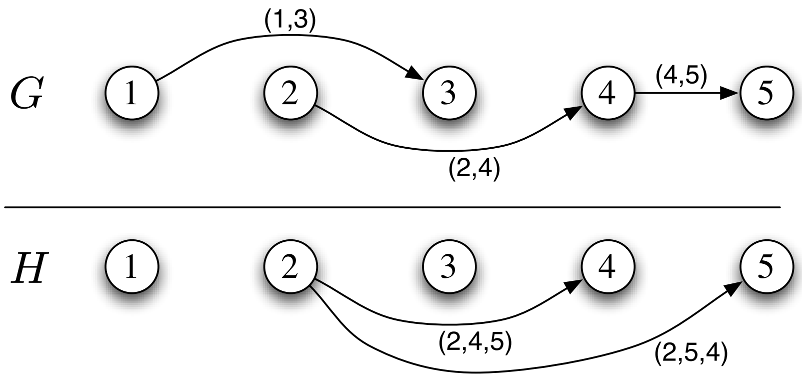


Figure 2.3: Branching structures for $G = \langle\{(1,3),(2,4),(4,5)\}\rangle$ and $H = \langle\{(2,4,5)\}\rangle$

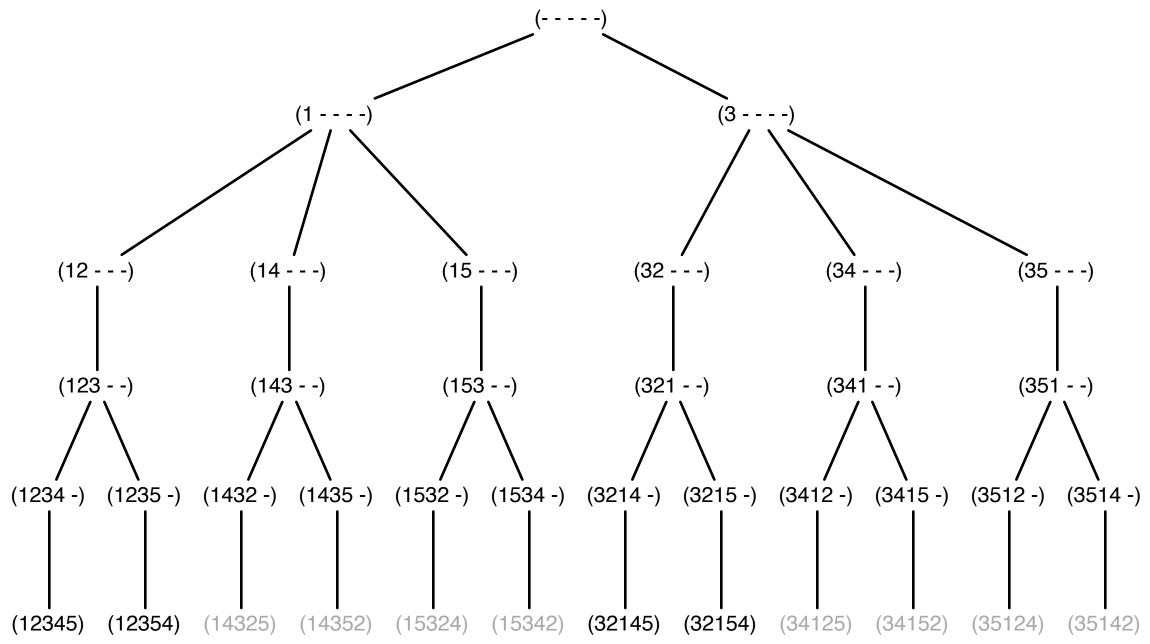


Figure 2.4: Search tree to compute $G : H$

The algorithm works as follows: the images of each point are initially considered unfinalized. Each recursive call to TRANSVERSAL-RECUR fixes the image of point j for the current (partial) permutation π (lines 8–9). When the permutation is finalized, it is added to set L if it is consistent with a topological ordering. Figure 2.4 shows a recursion tree with the current partial permutation at each step. The leaves set in black text form a right transversal. The permutation $\pi = (15324)$ is rejected because $2^\pi > 4^\pi$, and there is an edge from 2 to 4 in H's branching structure. Other leaves are rejected for similar reasons. This first version of the algorithm is somewhat inefficient—rather than continuing to the leaves, the branch $(1532\ \text{-})$ can be pruned because 2^π and 4^π have already been established.

Algorithm 4 incorporates this observation. Since $\tau_j^{-1}\tau_k$ stabilizes $\{1, 2, \ldots, j - 1\}$, the positions of points $1, 2, \ldots, j - 1$ are fixed, therefore if any of their positions are inconsistent with a topological ordering, the current recursion may be aborted. The relative positions of points $1, 2, \ldots, j - 2$ have already been checked, so each recursive step only performs a check against $j - 1$ (lines 1–3). The recursion tree in Figure 2.5 shows a slight improvement over that in Figure 2.4.

The algorithm can be improved further by analyzing the orbits of group G. The orbit of 2 is $\{2, 4, 5\}$. Therefore, at the branch $(15\ \text{-}\ \text{-}\ \text{-})$, we deduce that for any leaf π under

Algorithm 4 TRANSVERSAL-RECUR2(G, H, j, π)

```
1: if there is an edge from $i$ to $j-1$ in $H$ and $i^\pi > (j-1)^\pi$ then
2:     { $\pi$ is inconsistent with a topological ordering }
3:     return
4: else if $j = n$ then
5:     $L \leftarrow L \cup \{\pi\}$
6: else
7:     $\tau \leftarrow$ node labels from $G$
8:     for each $k$ reachable from $j$ in $G$ do
9:         TRANSVERSAL-RECUR2($G$, $H$, $j+1$, $\tau_j^{-1}\tau_k\pi$)
10:    end for
11: end if
```

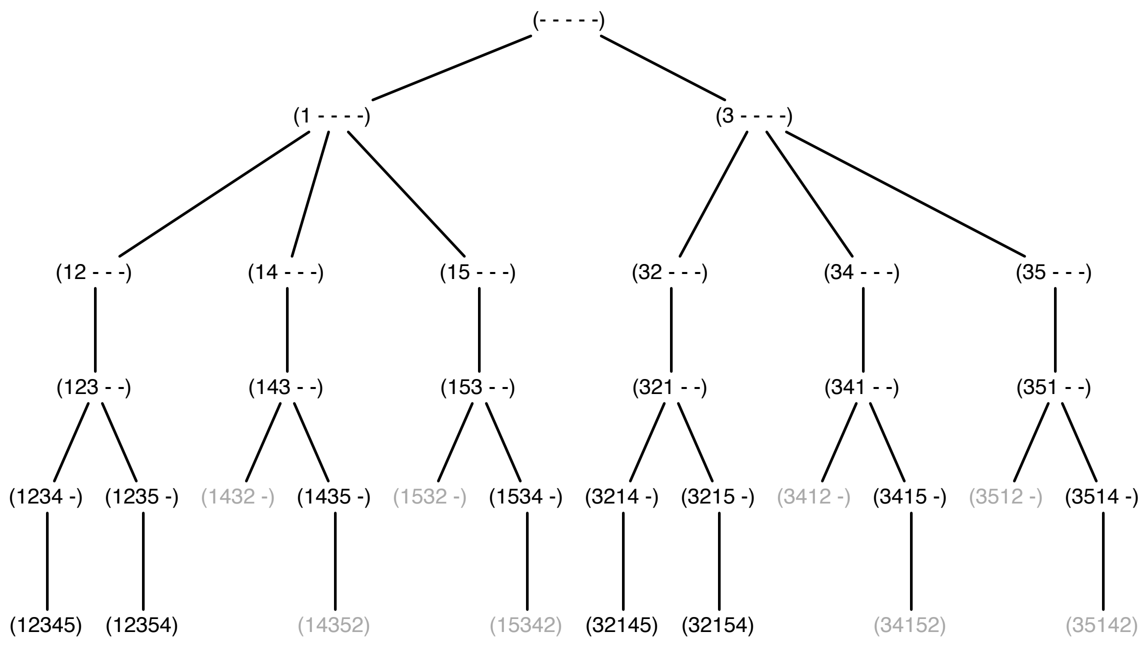


Figure 2.5: Search tree to compute $G : H$ with pruning

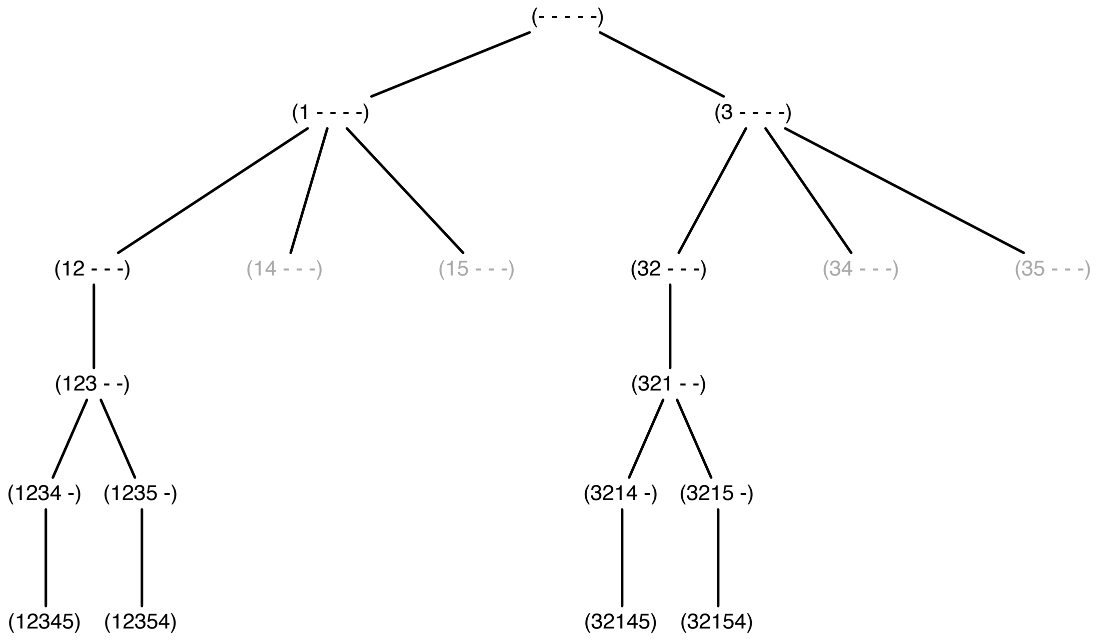


Figure 2.6: Search tree to compute $G : H$ with more efficient pruning

this branch, either $4^\pi = 2$ or $5^\pi = 2$. Both possibilities are invalid because either $2^\pi < 4^\pi$ or $2^\pi < 5^\pi$ will be violated. Figure 2.6 shows the recursion tree with further pruning.

2.4 From Points to Literals and Connections

Having covered the basics of permutation group theory, we are now ready to apply them to symmetries in functions and circuits. In these cases, permutations are not over *points*, but *Boolean literals*, requiring a slight change in notation. The next two chapters seek to answer the following questions:

1. Given a logical function, what permutations and/or negations of inputs and/or outputs leave the function unchanged?

2. Given a logical circuit, what permutations and/or negations of wires leave the circuit's function unchanged?

3. Functional Symmetries

Functional symmetries form the basis of this work. Given a function f, a **functional symmetry** is a permutation and/or negation of inputs and/or outputs that leaves the function unchanged.

In this chapter, we present a set of algorithms for computing the symmetries of a function given a set of truth tables or binary decision diagrams (BDDs). The next chapter addresses the situation where a clear-box logical *circuit* is given, rather than a black-box truth table. By *black-box*, we mean that a truth table presents only a mapping between inputs and outputs, whereas by *clear-box*, we mean that the internals of a circuit are visible.

Our approach is based on a reduction to graph isomorphism, rather than on finding symmetries directly. Greenspun's Tenth Rule of Programming states that any sufficiently complex C program contains a poorly-written Lisp interpreter.[1] A similar rule holds for us: any program which computes the symmetries of a Boolean function must reimplement one of the algorithms described in [McK81, Leo91]. Recent works [HK98, CS03, AP05, CK06] have all been based on the same fundamental approach. For this reason, and because implementations of the underlying algorithms are freely available [DLSM04, GAP06], we present our work within the framework of graph automorphism, with the understanding that the concepts can be applied to any of the preceding variants.

[1]"Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified bug-ridden slow implementation of half of Common Lisp.", http://philip.greenspun.com/research/

The backtrack algorithms described in [McK81, Leo91] (and their numerous derivatives) are based on the refinement of **ordered partitions**. Assuming that our permutations are defined over points, an initial partition is provided such that every point belonging to the same orbit also belongs to the same partition (the initial partition is an overapproximation of the orbits). The algorithms roughly consist of three steps which are repeated recursively:

1. Extend the current partial permutation, consisting of the sequence of partitions, with one element.
2. Make inferences, i.e., refine the ordered partition, in such a way that the refinement does not eliminate any valid permutations.
3. Block cosets from exploration.

The initial partition and refinement step are defined by the application, while the first and third steps are a generic portion of the algorithm. The refinement step is used to prevent the algorithm from searching for symmetry where none exists, so that we do not search all $n!$ permutations for symmetry. The last step uses the group axiom of closure to avoid searching for symmetries that can be deduced from other known symmetries; if $(1,2)$ is a symmetry and $(2,3)$ is a symmetry, there is no need to check for $(1,3)$.

A Note on Notation. Group theory and graph theory use similar notation to refer to different concepts. Therefore, we let G (set in standard face) denote a permutation group, and $\mathfrak{G}$ (set in fraktur face) denote a graph (V, E). Rather than the points $\{1, 2, \ldots, n\}$, we allow permutations to be defined over other domains, such as the set of nodes V. To avoid confusion, we will write permutations in positional notation, so that for any $u, v \in V$, the tuple (u, v) denotes an edge (*not* a permutation). Then for some $\pi \in G$, we may define the image $\mathfrak{G}^\pi$ of $\mathfrak{G} = (V, E)$ as follows:

$$\mathfrak{G}^\pi = (V, \{(u^\pi, v^\pi) : (u, v) \in E\})$$

Definition. A **graph isomorphism** $\alpha : V \to V'$ between graphs (V, E) and (V', E') is a bijection of the nodes that preserves edge relationships: $(u, v) \in E \iff (\alpha(u), \alpha(v)) \in E'$. For colored graphs (V, E, c) where $c : V \to \mathbb{N}$, an isomorphism must also preserve colors: $c(v) = c'(\alpha(v))$. A **graph automorphism** is an isomorphism between a graph and itself.

To use the terminology of [Leo91], graph automorphism is a "subgroup-type" problem, that is, given a graph $\mathfrak{G} = (V, E)$ of n nodes and a predicate $Aut(\pi, \mathfrak{G}) \mapsto \mathfrak{G} = \mathfrak{G}^{\pi}$, the algorithm finds the largest subgroup $G \subseteq S_n$ such that $Aut(\pi, \mathfrak{G}), \forall \pi \in G$. Suppose $p : V \rightarrow \mathbb{N}$ describes the current partition. The initial partition divides the nodes according to their degree:

$$p(a) = p(b) \equiv |\{v : (a, v) \in E\}| = |\{v : (b, v) \in E\}| \tag{3.1}$$

The refinement step from [McK81] works as follows: given nodes a and b, if their neighbors are in different partitions, then so are they in the refined partition p':

$$p'(a) = p'(b) \equiv p(a) = p(b) \wedge \{p(v) : (a, v) \in E\} = \{p(v) : (b, v) \in E\} \tag{3.2}$$

This is a consequence of the definition of graph automorphism:

$$(a, b) \in E \equiv (a^{\pi}, b^{\pi}) \in E, \forall \pi \in G \tag{3.3}$$

Other properties that are invariant under isomorphism may be found [Lev74], and lead to other possible refinement steps.

Given the task of finding the symmetries of a Boolean function, we may define an initial partition and refinement step in a similar fashion. We described potential initial partitioning and refinement steps in [CK06]. The translation from these to equivalent steps in the graph domain is the subject of the remainder of this chapter. For the remainder, let n and m indicate the number of inputs and outputs to a function, respectively. Inputs are denoted by $x_i \in \mathbb{B}$, or simply $x \in \mathbb{B}^n$. Outputs are denoted by $f_j : \mathbb{B}^n \rightarrow \mathbb{B}$ or $y_j \in \mathbb{B}$ depending on context. A **cube** is a conjunction of variables or alternatively the convex subset of $\mathbb{B}^n$ for which the conjunction is true. A cube may be written as a set, a conjunction, or a bitstring with dashes, e.g. $\{x_3, \bar{x}_2, x_0\}$, $x_3\bar{x}_2x_0$, and 10-1 are equivalent. A **minterm** is a single point in $\mathbb{B}^n$. For a function f, the **satisfy count** denoted by $|f|$ is $|\{x : f(x) = 1\}|$. The **cofactor** of f with respect to some literal x_i ($\bar{x}_i$), denoted by f_{x_i} ($f_{\bar{x}_i}$), is an n-input function equivalent to f but with x_i substituted by 1 (0), e.g.

$$f_{x_i}(x) = f(x_{n-1}, \ldots, x_{i+1}, 1, x_{i-1}, \ldots, x_0)$$

The cofactor of f with respect to a cube is similar, e.g.

$$f_{x_i\bar{x}_{i-1}}(x) = f(x_{n-1}, \ldots, x_{i+1}, 1, 0, x_{i-2}, \ldots, x_0)$$

An **implicant** of a function is a cube for which the function evaluates to true. A **prime implicant** (or simply prime) is one which is not contained in any other implicants. For example, if $f = x_1 + x_0$, then $x_1 x_0$ is an implicant of f and contained in the prime x_1.

We use the term **move** as before, to describe the action of permutations. We say that a point i can be moved to point j if there exists a symmetry π such that $i^\pi = j$.

3.1 Graph Formulation

To find the symmetries of a function $f \colon \mathbb{B}^n \to \mathbb{B}^m$ via graph automorphism, we will define a mapping from f to a colored graph $(V_X \cup V_Y \cup V_M, E, c)$. V_X consists of $2n$ nodes to represent input literals $x_i, \bar{x}_i$. V_Y consists of $2m$ nodes to represent output literals $y_j, \bar{y}_j$. A permutation and/or negation of inputs and/or outputs of a function f is represented by a permutation π over $V_X \cup V_Y$; the new function is denoted by f^π. What remains is to define V_M, E, and c so that the graph's automorphisms correspond exactly to the function's symmetries. In other words, V_M, E, and c will uniquely encode f. It is understood that when we say an automorphism is a functional symmetry, we consider only the mapping over $V_X \cup V_Y$; similarly, when we say a functional symmetry is an automorphism, we mean that it can be suitably extended to nodes in V_M.

Let $Sym(f)$ denote the symmetries of f, and $Aut(\mathfrak{G})$ denote the automorphisms of $\mathfrak{G}$. The graph formulations in this chapter satisfy the criteria in the following theorem:

Theorem 1. *Suppose Γ maps a function f to a colored graph $\mathfrak{G} = (V_X \cup V_Y \cup V_M, E, c)$ with the following criteria:*

1. *V_X and V_Y are defined as above.*
2. *For any $u, v \in V_X$, $(u, v) \in E$ iff $u = \bar{v}$ and for any $u, v \in V_Y$, $(u, v) \in E$ iff $u = \bar{v}$.*
3. *$c(v)$ is 0 if $v \in V_X$, 1 if $v \in V_Y$, and some number ≥ 2 otherwise.*
4. *Γ is an bijection, i.e. $f = f'$ iff $\Gamma(f) = \Gamma(f')$ (modulo relabeling of nodes in V_M). Formally, $f = f'$ iff there exists an isomorphism ϕ between $\Gamma(f)$ and $\Gamma(f')$ where $\forall l \in V_X \cup V_Y, \phi(l) = l$.*
5. *For any π, there exists an isomorphism ϕ between $\Gamma(f^\pi)$ and $\Gamma(f)$ where $\forall l \in V_X \cup V_Y, \pi(l) = \phi(l)$.*

$$\begin{tikzcd}
f \arrow[rr] \arrow[d, leftrightarrow, "(4)"] & \pi \arrow[d, leftrightarrow, "(5)"] & f^{\pi} \arrow[d, leftrightarrow, "(4)"] \\
\Gamma(f) \arrow[rr] & \phi & \Gamma(f^{\pi})
\end{tikzcd}$$

Figure 3.1: Diamond property required by Theorem 1.

Then $\alpha \in Sym(f) \iff \alpha \in Aut(\Gamma(f))$.

Proof. ($\Longrightarrow$) follows trivially from criteria 1 and 4. ($\Longleftarrow$) criteria 2 and 3 ensure that permutations of literals correspond to valid permutations/negations of inputs/outputs, and criteria 4 and 5 ensure that an isomorphism ϕ between $\Gamma(f)$ and $\Gamma(f')$ implies $f^{\phi} = f'$. $\square$

Criterion 2 enforces Boolean consistency, e.g. $\overline{\alpha(x_i)} = \alpha(\bar{x}_i)$. Criteria 4 excludes the degenerate case where V_M consists of a single node with a unique label or color. Criterion 5 excludes the case where Γ encodes f in $|V_M|$. Another way of restating the criteria is that Γ is information preserving, or that Γ treats all literals identically. Figure 3.1 illustrates the so-called diamond property enforced by criteria 4 and 5.

3.1.1 The Trivial Graph Formulation

Let f be a multiple-output function for which $f_j \colon \mathbb{B}^n \to \mathbb{B}$ describes the function of the j-th output. In the naïve graph formulation, V_M consists of 2^n nodes, each synonymous with a minterm in $\mathbb{B}^n$. An edge connects node $u \in V_X$ with $v \in V_M$ iff the literal u appears in minterm v. An edge connects connects $v \in V_M$ with $y_j \in V_Y$ iff $f_j(v) = 1$, otherwise v is connected to $\bar{y}_j$.

It follows directly that any automorphism of $\mathfrak{G}$ is a symmetry of f. Luks [Luk99] provides an equally simple reduction to hypergraph automorphism, and Kisielewicz [Kis98] provides a simple reduction from graph automorphism to the symmetry problem for Boolean functions.

Example 3.1. Suppose we are given the single-output function $f_0 = x_2(\bar{x}_1 + x_0)$. V_X consists of six nodes $\{x_2, \bar{x}_2, x_1, \bar{x}_1, x_0, \bar{x}_0\}$, V_Y consists of two nodes, and V_M consists of eight nodes. Figure 3.2a shows the graph formulation for f_0. Labels and line styles are not part of the formulation, but rather are visual aids for the reader.

A few remarks should be repeated. First, the exponential size of $\mathfrak{G}$ ($|V_M| = 2^n$) is likely to be problematic for runtime and memory. Second, the refinement rule for graph automorphism is based on one of many invariants that hold for automorphisms of $\mathfrak{G}$. As the continuation of our example shows, several invocations of this refinement rule may be required to equal one invocation of one of the application-specific rules from [CK06].

Example 3.2. In the function $f = x_2(\bar{x}_1 + x_0)$, variable x_2 cannot be moved to x_0 without changing f. This follows from the fact that $|f_{x_2}| \neq |f_{x_0}|$, or if we consider negations,

$$\{|f_{x_2}|, |f_{\bar{x}_2}|, |\bar{f}_{x_2}|, |\bar{f}_{\bar{x}_2}|\} \neq \{|f_{x_0}|, |f_{\bar{x}_0}|, |\bar{f}_{x_0}|, |\bar{f}_{\bar{x}_0}|\}$$

All of the satisfy counts may be computed from a BDD for f, in time which is polynomial with respect to the size of the BDD. This same fact can be established by performing several refinement steps on Figure 3.2a. We first partition the nodes by color:

$$(V_X, V_Y, V_M)$$

then by degree:

$$(V_X, \{y_0\}, \{\bar{y}_0\}, V_M)$$

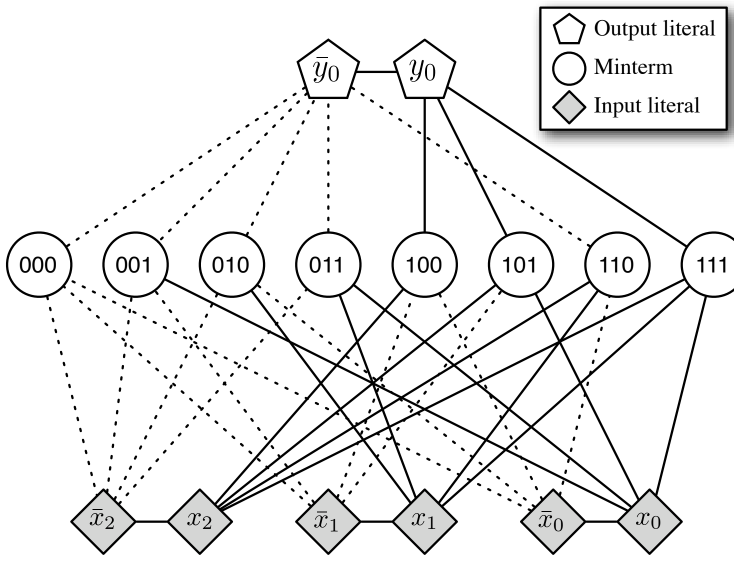
The splitting of V_Y causes V_M to split via the refinement step:

$$(V_X, \{y_0\}, \{\bar{y}_0\}, \{100, 101, 111\}, \{000, 001, 010, 011, 110\})$$

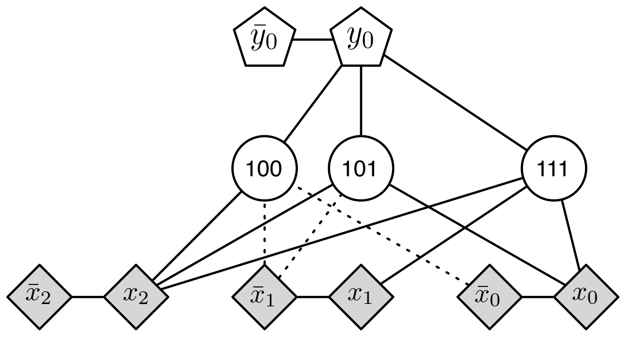
This in turn, causes V_X to split when we count for each node in V_X its neighbors in $\{100, 101, 111\}$ and $\{000, 001, 010, 011, 110\}$. It is not a coincidence that these are the satisfy counts. The difference is in runtime: each refinement step on the graph takes $\mathcal{O}(2^n)$ time (since $|V_M| = 2^n$), while the runtime to compute satisfy counts would usually be polynomial in n given a BDD.

3.2 Improved Graph Formulation for Single Outputs

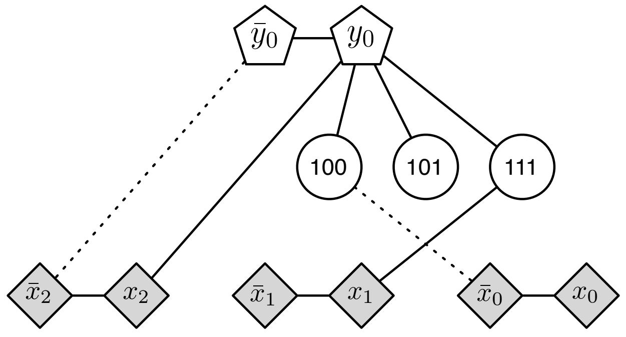
Let us assume for now that f is a function with a single output. We present three improvements to the graph formulation which reduce the size of $\mathfrak{G}$ and/or expose more



(a)



(b)



(c)

Figure 3.2: Graph formulations for $f_0 = x_2(\bar{x}_1 + x_0)$.

information to a graph automorphism solver.

3.2.1 Reducing $|V_M|$ with Satisfy Counts

In the naïve formulation, V_M contains a node for each minterm in $\mathbb{B}^n$ in order to represent symmetries which negate the output. Suppose that we are computing symmetries over the inputs *only* and do not want symmetries which negate the output. That is, we seek those π such that $f = f^\pi$ and $y^\pi = y$. Since the codomain of f is $\mathbb{B}$, $f = f^\pi$ if and only if $\bar{f} = (\bar{f})^\pi$. Then there is no need to introduce both minterm nodes for which $f(x) = 1$ as well as minterm nodes for which $f(x) = 0$; we can arbitrarily choose one of the two sets without adding or removing automorphisms from $\mathfrak{G}$.

If we are interested in finding all symmetries, but know *a priori* that no symmetries negate the output, we can again safely remove one of the two sets of minterms. The satisfy counts are an efficient means to determine this fact: if $|f| \neq |\bar{f}|$, then no symmetry moves y to $\bar{y}$. We codify this into the following rule:

- if $|f| < |\bar{f}|$, remove nodes corresponding to minterms in $\bar{f}$
- if $|f| > |\bar{f}|$, remove nodes corresponding to minterms in f

For the degenerate case where $f = 0$ $(\bar{f} = 0)$ we remove V_M and mark y $(\bar{y})$ with a new color. The result of applying this rule on Figure 3.2a is shown in Figure 3.2b.

We note that in contrast, the construction proposed in [CMB05a] colored the 2^n minterm nodes using two colors for the on- and off-set.[2] This forces the graph automorphism solver to unnecessarily compute symmetries for both f and $\bar{f}$.

3.2.2 Reducing $|E|$ with Satisfy Counts

As in the previous case, we may reduce the size of $\mathfrak{G}$ if we can determine *a priori* that certain symmetries are not possible. If $|f_{x_i}| \neq |f_{\bar{x}_i}|$, then x_i cannot be moved to $\bar{x}_i$, and the edges from x_i (or $\bar{x}_i$) to nodes in V_M may be removed. In order to preserve all automorphisms of $\mathfrak{G}$ we need to apply any rule consistently for each input. We use the following:

- if $|f_{x_i}| < |f_{\bar{x}_i}|$, remove edges from $\bar{x}_i$ to nodes in V_M. Additionally, if $|f_{x_i}| = 0$, connect $\bar{x}_i$ to y and x_i to $\bar{y}$.

[2]For an m-output function, 2^m colors are used.

- if $|f_{x_i}| > |f_{\bar{x}_i}|$, remove edges from x_i to nodes in V_M. Additionally, if $|f_{\bar{x}_i}| = 0$, connect x_i to y and $\bar{x}_i$ to $\bar{y}$.

The case where $|f_{x_i}| = 0$ is a generalization of the case where $f = 0$. Note that the rule removes edges from $\bar{x}_i$ in one case and x_i in the other. This is necessary in case some x_i may be moved to some $\bar{x}_j$, as in the following example.

Example 3.3. Let $f = x_2(\bar{x}_1 + x_0)$. If we compare the satisfy counts of the various cofactors, $|f_{\bar{x}_2}| < |f_{x_2}|$, $|f_{x_1}| < |f_{\bar{x}_1}|$ and $|f_{\bar{x}_0}| < |f_{x_0}|$. Therefore, node $x_2\bar{x}_1\bar{x}_0y$ is replaced by node $\bar{x}_0y$, $x_2\bar{x}_1x_0y$ by y, and node $x_2x_1x_0$ is replaced by x_1y. Node x_2 is connected to y_0 since $|f_{\bar{x}_2}| = 0$. The result is shown in Figure 3.2c. Note that swapping $\bar{x}_1$ with x_0 is a symmetry of f, and the new graph preserves the corresponding automorphism.

3.2.3 Reducing $|V_M|$ with Unateness

Rather than represent a function f using minterms, we can use any representation which is unique and which does not depend on an ordering of the variables. The set of all minterms is unique, but large. A BDD for a function is often small, but depends on an ordering of the variables. A sum-of-products is often small, but not unique unless if we include implicants based only on some variable-agnostic property, for example:

- the set of *all* implicants
- the set of all implicants with n literals (minterms)
- the set of all implicants with at least k literals
- the set of all prime implicants
- the set of all essential prime implicants plus those minterms not covered by the essential primes. A prime is essential if it contains a minterm that is not contained in any other prime.

These sets all satisfy Theorem 1; e.g. for any f and f^π, their respective sets of primes are isomorphic. These sets are large for the general class of Boolean functions, but small for a very common subclass (unate functions).

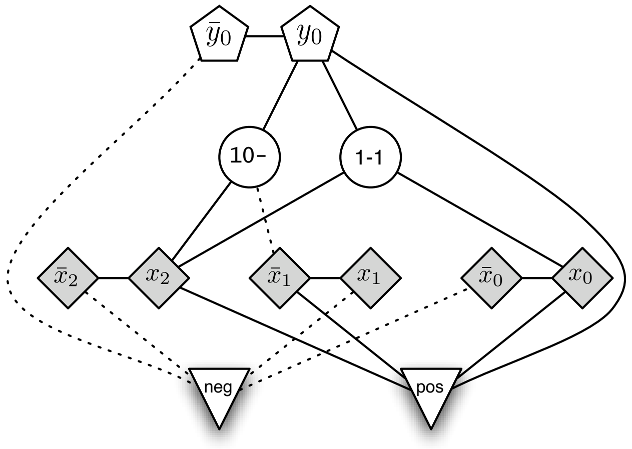


Figure 3.3: Graph containing primes of $x_2(\bar{x}_1 + x_0)$.

Definition. A function f is positive (negative) **unate** in x_i if $f_{x_i} \supseteq f_{\bar{x}_i}$ ($f_{x_i} \subseteq f_{\bar{x}_i}$); when f is understood from context, we refer to the variables themselves as unate. A variable is **binate** if it is not unate. A function is unate if it is unate in all variables.

It is well-known that all primes of a unate function are essential, and therefore this set is small [BHMSV84]. Furthermore if f is unate, then so is $\bar{f}$. Then any unate function f can be represented by the primes of f and $\bar{f}$, for an exponential decrease in graph size compared to enumeration of minterms.

Example 3.4. Let $f = x_2(\bar{x}_1 + x_0)$. The primes of f are $\{x_2\bar{x}_1, x_2x_0\}$ and the primes of $\bar{f}$ are $\{\bar{x}_2, x_1\bar{x}_1\}$ for a total of 4 nodes in V_M. In comparison, there are 8 minterms in $\mathbb{B}^3$.

We modify our formulation as follows. Two new nodes of color 3 are introduced to mark each literal as being positive or negative unate in f. A special property of unate functions is that if f is positive (negative) unate in x, then $\bar{f}$ is negative (positive) unate in x. To preserve symmetry, the node denoting positive unateness is adjacent to the node labeled "f", and the node denoting negative unateness is adjacent to the node labeled "$\bar{f}$". The resulting graph for $f = x_2(\bar{x}_1 + x_0)$ is shown in Figure 3.3. Since $|f| < |\bar{f}|$, only f is represented.

Suppose a function f is unate in only a subset of its variables, say X_U and binate in variables $X_B = X \setminus X_U$. Then f can be uniquely written as $2^{|X_B|}$ sets of primes, one for

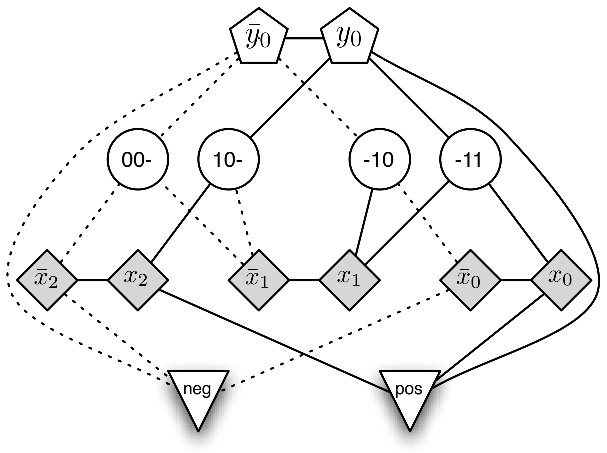


Figure 3.4: Graph containing primes of $\bar{x}_1 x_2 + x_1 x_0$.

each cofactor of f with respect to the binate variables. Thus the function

$$f = (x_0 \oplus x_1)x_2 + (x_0 + x_1)x_3$$

which is unate in x_2 and x_3, is uniquely represented by the five cubes

$$\begin{aligned} f &= x_0\bar{x}_1 f_{x_0\bar{x}_1} + \bar{x}_0 x_1 f_{\bar{x}_0 x_1} + x_0 x_1 f_{x_0 x_1} \\ &= x_0\bar{x}_1(x_2 + x_3) + \bar{x}_0 x_1(x_2 + x_3) + x_0 x_1(x_3) \\ &= x_0\bar{x}_1 x_2 + x_0\bar{x}_1 x_3 + \bar{x}_0 x_1 x_2 + \bar{x}_0 x_1 x_3 + x_0 x_1 x_3 \end{aligned}$$

For this case, the binate variables are obviously not connected to either of the nodes marking unateness.

Example 3.5. Let $f = \bar{x}_1 x_2 + x_1 x_0$. Since $|f| = |\bar{f}|$, both f and $\bar{f}$ need to be represented in any graph formulation. The primes of f ($\bar{x}_1 x_2 + x_1 x_0$) and $\bar{f}$ ($\bar{x}_1\bar{x}_2 + x_1\bar{x}_0$) are shown in Figure 3.4. Note that y_0 can be moved to $\bar{y}_0$ iff the unate variables are simultaneously negated.

3.2.4 Combining Transformations

One may notice that it is not strictly necessary to mark variables as unate or not in the previous examples—the connectivities of the graphs already show this. However, doing

Algorithm 5 GRAPH-REDUCTION(f)

```
1:  $X_P \leftarrow$ positive unate variables of $f$
2:  $X_N \leftarrow$ negative unate variables of $f$
3:  $X_B \leftarrow$ binate variables of $f$
4:  connect nodes in $X_P$ to "pos" node
5:  connect nodes in $X_N$ to "neg" node
6:  add edge ("pos", $y$)
7:  add edge ("neg", $\bar{y}$)
8:  **if** $|f| \leq |\bar{f}|$ **then**
9:      **for each** assignment $c$ to $X_B$ variables **do**
10:         **for each** prime implicant $p$ of $f_c$ **do**
11:             add node $pc$ to $V_M$
12:             add edge $(pc, y)$
13:             **for each** literal $l_i$ in cube $p$ **do**
14:                 add edge $(l_i, pc)$
15:             **end for**
16:             **for each** literal $l_i$ in cube $c$ **do**
17:                 **if** $|f_{l_i}| \leq |f_{\bar{l}_i}|$ **then**
18:                     add edge $(l_i, pc)$
19:                 **end if**
20:             **end for**
21:         **end for**
22:     **end for**
23: **end if**
24: **if** $|f| \geq |\bar{f}|$ **then**
25:     do same as above, but with $\bar{f}$ and $\bar{y}$ instead of $f$ and $y$
26: **end if**
```

so allows us to correctly apply other transformations without destroying this information. Suppose x_i is binate, and $|f| < |\bar{f}|$ and $|f_{x_i}| < |f_{\bar{x}_i}|$. If we were to remove all edges from $\bar{x}_i$, it would no longer be distinguishable from the unate variables. Algorithm 5 illustrates the combination of the previous three techniques.

Note that the unateness transformation exposes some differences between input variables and obscures others. Before this transformation, the degree of node x_i is $|f_{x_i}| + 1$, assuming $|f| < |\bar{f}|$ and $|f_{x_i}| < |f_{\bar{x}_i}|$. This means that the graph automorphism refinement procedure will distinguish two variables x_i and x_j if $|f_{x_i}| \neq |f_{x_j}|$ since the degrees of the respective nodes will also differ (see Equation 3.1). This does not necessarily hold if we use prime implicants to represent f. The following example adapted from [CK06] illustrates this in detail.

x_5	x_4	x_3	x_2	x_1	x_0
0	0	0	1	1	1
0	1	1	0	0	1
0	1	1	0	1	1
0	0	1	0	1	1
1	1	1	1	1	1
1	0	0	1	1	0
$\langle b,2,?\rangle$	$\langle b,3,?\rangle$	$\langle b,4,?\rangle$	$\langle b,3,?\rangle$	$\langle p,5,?\rangle$	$\langle b,5,?\rangle$

(a)

x_5	x_4	x_3	x_2	x_1	x_0
0	0	0	1	1	1
0	1	1	0	-	1
0	0	1	0	1	1
1	1	1	1	1	1
1	0	0	1	1	0
$\langle b,2,2\rangle$	$\langle b,3,2\rangle$	$\langle b,4,3\rangle$	$\langle b,3,3\rangle$	$\langle p,5,4\rangle$	$\langle b,5,4\rangle$

(b)

x_1	x_5	x_4	x_2	x_3	x_0
1	0	0	1	0	1
-	0	1	0	1	1
1	0	0	0	1	1
1	1	1	1	1	1
1	1	0	1	0	0
$\langle p,5,4\rangle$	$\langle b,2,2\rangle$	$\langle b,3,2\rangle$	$\langle b,3,3\rangle$	$\langle b,4,3\rangle$	$\langle b,5,4\rangle$

(c)

Figure 3.5: Refining variable partitions of $f = x_5\bar{x}_4\bar{x}_3x_2x_1\bar{x}_0 + (\bar{x}_5\bar{x}_4\bar{x}_3 + x_5x_4x_3)x_2x_1x_0 + (x_4 + x_1)\bar{x}_5x_3\bar{x}_2x_0$ using an implicant table.

Example 3.6. Let $f = x_5\bar{x}_4\bar{x}_3x_2x_1\bar{x}_0 + (\bar{x}_5\bar{x}_4\bar{x}_3 + x_5x_4x_3)x_2x_1x_0 + (x_4 + x_1)\bar{x}_5x_3\bar{x}_2x_0$. The function contains no symmetry, and we would like to confirm this using a minimal amount of branch and bound. The table in Figure 3.5a shows the six minterm implicants in f; since $|f| = 6 < |\bar{f}| = 26$, $\bar{f}$ is not represented. Under each column for x_i is a triple denoting whether x_i is binate and $|f_{x_i}|$. From this information, we can deduce that x_5 cannot be moved to x_4. Other facts cannot be deduced: we do not know whether x_4 can be moved to $\bar{x}_4$ since $|f_{x_4}| = |f_{\bar{x}_4}| = 3$.

Since x_1 is unate, the implicants are rewritten as the entries in Figure 3.5b. The last entry in each triple is the number of 1s in each column, i.e. the number of times literal x_i appears in the implicants. Since a 2 appears for x_4, and $2 \neq 3$, we deduce that x_4 cannot be moved to $\bar{x}_4$. However, a 2 appears for both x_5 and x_4, meaning that the degrees of their respective nodes are now equal and thus the variables cannot be distinguished anymore using only a single application of Equation 3.1. In order to facilitate the graph automorphism solver, it may be beneficial (but not necessary) to provide this information as an extension to the graph formulation.

3.3 Multiple-output Functions

Suppose f consists of more than one output ($m > 1$). Then the graph formulation is simply the composition of the formulations for each f_j, where the $2n$ nodes for V_X are shared. This explains why unate variables are marked as such by connecting to special nodes—different outputs are unate for different variables. Any nodes in V_M with identical neighbors in V_X may also be merged, so that the outputs are defined over a shared space of at most 3^n nodes.

Definition. Given a multiple output function $(f_0(x), f_1(x), \ldots, f_{m-1}(x))$, the **characteristic function** χ is defined such that $\chi(x, y_0, y_1, \ldots, y_{m-1}) = \bigwedge_{i=0}^{m-1}(f_i(x) \equiv y_i)$.

An alternative is to analyze the characteristic function of f. The symmetries of f can then be obtained by analyzing χ, where we consider all negations and permutations of inputs x_i and y_i for which an x input is not swapped with a y input.

3.4 Experimental Results

To assess the relative improvement realized by various encodings, we compare different graph encodings on a variety of functions. All experiments were performed on a Pentium4 2GHz computer running Linux with the `perfctr` library[3] for cycle-accurate timing measurement, CUDD[4] for BDD manipulation, and Saucy [DLSM04] for finding graph automorphisms. Reported runtimes include all steps necessary to find all symmetries of a multi-output function expressed as m BDDs. Each formulation was verified to produce the correct symmetry group.

The first set of experiments tests whether redundant nodes and edges are useful in the refinement steps of the Saucy solver, or whether they unnecessarily add to runtime. We extracted single-output functions from two mapped circuits from the IWLS 2005 benchmark suite and computed their symmetries using three graph formulations: the basic formulation, the formulation reduced with §3.2.1, and the formulation reduced with §3.2.1 and §3.2.2. The results are in Table 3.1. Column n denotes the number of inputs, while # denotes the number of functions examined. “Time”, “$|V|$”, and “$|E|$” respectively denote average runtime in milliseconds, vertices in graph, and edges in graph. As the results indicate, the proposed optimizations greatly reduce the runtime and memory requirements for symmetry detection.

[3]http://user.it.uu.se/~{}mikpe/linux/perfctr/
[4]http://vlsi.colorado.edu/~fabio/CUDD/

		Basic			Using §3.2.1			Using §3.2.1 and §3.2.2		
n	#	Time	$\|V\|$	$\|E\|$	Time	$\|V\|$	$\|E\|$	Time	$\|V\|$	$\|E\|$
3	10259	0.053	16	36	0.058	12.9	23.7	0.053	12.9	21.3
4	8433	0.105	26	85	0.087	17.5	42.7	0.075	17.5	37.7
5	9017	0.254	44	198	0.198	28.0	102	0.149	28.0	85.8
6	4800	0.981	78	455	1.167	46.9	237	0.735	46.9	191
7	1743	5.383	144	1032	2.743	58.5	348	1.667	58.5	273
8	806	2.501	274	2313	1.070	92.9	683	0.669	92.9	504
9	344	5.320	532	5130	1.204	141	1218	0.685	141	837

Table 3.1: Single-output functions.

The second set of experiments compares three graph formulations for multiple-output functions. Each benchmark circuit in Table 3.2 is analyzed as a *single* multiple-output function, e.g. pcle is a 19-input 9-output function. Formulation "χ" represents the trivial formulation which treats each circuit using its characteristic function. The next set of columns represents this formulation after applying the rules in §3.2.2–§3.2.3. The last set of columns represents the formulation which treats each output independently and simply composes the graphs for each output. The results indicate that for binate functions, e.g. alu4, the trivial formulation may be best by a small margin. However, in most cases, the formulation which composes the graphs corresponding to each output is best by a large margin.

3.5 Previous Work

A number of previous works have addressed the problem of finding symmetries of Boolean functions, and all complete algorithms[5] rely on the same basic ideas set forth in [McK81] and [Leo91]. We can divide these algorithms into two sets: those based on comparison of satisfy counts (spectral methods) and those based on comparison of output values at various input minterms. Both families of algorithms are targeted towards finding symmetries of single-output functions, as we explain below. For simplicity, let us assume that we are searching only for symmetries among the $n!m!$ permutations of inputs and outputs, and disregard negation.

[5]That is, algorithms which are able to find *all* symmetries.

			χ			χ with §3.2.1–§3.2.3			§3.3 with §3.2.1–§3.2.3		
Benchmark	n	m	Time	$\|V\|$	$\|E\|$	Time	$\|V\|$	$\|E\|$	Time	$\|V\|$	$\|E\|$
alu4	14	25	**134.972**	16428	360470	231.143	16460	299316	151.63	23816	249456
b12	15	9	225.293	32816	786456	113.095	13196	213486	**2.0009**	130	549
cm163a	16	5	672.044	65578	1.37628e6	359.228	27840	495725	**1.6644**	233	1547
cu	14	11	203.448	16434	409625	97.1851	6366	99795	**2.0757**	92	301
misex3	14	14	126.089	16440	458780	**87.073**	7364	121142	90.801	15074	146789
pcle	19	9	8585.57	524344	1.46801e7	2001.99	124475	2.740e6	**4.7199**	839	7542
pm1	16	13	2255.81	65594	1.90057e6	3349.99	65646	1.287e6	**1.6272**	116	327
sct	19	15	14258.6	524356	1.78258e7	8669.96	524416	1.179e7	**8.5526**	634	5690
spla	16	46	1761.13	65660	4.06329e6	486.632	22225	405676	**107.92**	9918	94780
table3	14	14	139.165	16440	458780	**61.5105**	3383	56637	64.644	8684	82788
table5	17	15	2438.34	131136	4.19434e6	**473.38**	28792	597728	672.80	105836	1.142e6
tcon	17	16	3.893e6	131138	4.32541e6	23517.2	131202	3.8012e6	**1.4030**	146	241
vda	17	39	5414.17	131184	7.34009e6	957.751	42667	798412	**141.51**	6931	49772

Table 3.2: MCNC91 benchmarks.

3.5.1 Spectral Methods

Algorithms such as [AP05, Wan06, ABPS07] try to reduce the complexity of a function by abstracting the function's 2^n minterms into a small set of **signatures** based on various satisfy counts. To recap, the satisfy count of a function f, denoted by $|f|$, is $|\{x|f(x) = 1\}|$. If two variables have differing signatures, for example if $|f_{x_0}| \neq |f_{x_1}|$, then x_0 cannot be moved to x_1.

Spectral techniques are fast only when the signatures are effective at distinguishing variables, but as the authors of [MMM95] show, signatures are not particularly effective at doing so. As an illustration, the algorithm proposed in [AP05] performs partition refinement using the $(n^2 + n)/2$ signatures of the form $|f_{x_i}|$ and $|f_{x_i x_j}|$, and no others. Let us define the order of a signature as the number of literals in the cube with which we are cofactoring, i.e. $|f|$ is a zeroth order signature, $|f_{x_0}|$ is a first order signature, and $|f_{x_0 x_1}|$ is a second order signature. First order signatures $|f_{x_i}|$ are used for an initial partitioning analogous to Equation 3.1, and second order signatures $|f_{x_i x_j}|$ are used for refinement analogous to Equation 3.2.

It is entirely likely that two variables x_i and x_j have the same signatures, i.e. $|f_{x_i}| = |f_{x_j}|$ and $|f_{x_i x_k}| = |f_{x_j x_k}|$, yet x_i may not be moved to x_j. In this case, the algorithm in [AP05] would need to exhaustively search through a large set of permutations to find the few symmetries of f. The next example illustrates this point.

Example 3.7. Let us suppose that we are searching for the symmetries of the two output function described by:

$$f_0 = x_2 + x_1 + x_0$$
$$f_1 = x_1 \oplus x_0$$

Since none of the algorithms work directly with multiple-output functions, we define the single-output characteristic function $\chi(x, y) = (f_0(x) \equiv y_0) \wedge (f_1(x) \equiv y_1)$. The following truth table for χ shows the rows for which $\chi(x, y) = 1$.

x_2	x_1	x_0	y_1	y_0	χ
0	0	0	0	0	1
0	0	1	1	1	1
0	1	0	1	1	1
0	1	1	0	1	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	0	1	1
		otherwise			0

x_2	x_1	x_0	y_1	y_0	χ
0	0	0	0	0	1
0	0	**1**	**1**	1	1
0	1	0	1	1	1
0	1	1	0	1	1
1	0	0	0	1	1
1	0	**1**	**1**	1	1
1	1	0	1	1	1
1	1	1	0	1	1

(a) $|\chi_{y_1 x_0}| = 2$

x_2	x_1	x_0	y_1	y_0	χ
0	0	0	0	0	1
0	0	1	1	1	1
0	**1**	0	**1**	1	1
0	1	1	0	1	1
1	0	0	0	1	1
1	0	1	1	1	1
1	**1**	0	**1**	1	1
1	1	1	0	1	1

(b) $|\chi_{y_1 x_1}| = 2$

x_2	x_1	x_0	y_1	y_0	χ
0	0	0	0	0	1
0	0	1	1	1	1
0	1	0	1	1	1
0	1	1	0	1	1
1	0	0	0	1	1
1	0	1	**1**	1	1
1	1	0	**1**	1	1
1	1	1	0	1	1

(c) $|\chi_{y_1 x_2}| = 2$

Figure 3.6: Signatures of $\chi = ((x_2 + x_1 + x_0) \equiv y_0) \wedge ((x_1 \oplus x_2) \equiv y_1)$.

By expressing f as a characteristic function, we increase the order of the signatures by one, e.g. $|f_0| = |\chi_{y_0}|$ and $|(f_0)_{x_0}| = |\chi_{y_0 x_0}|$. Second order signatures that would be used if analyzing f_0 or f_1 in isolation are not used, because they are now third order signatures of χ. Many first and second order signatures of χ are equal by construction: $|\chi_{x_i}| = 2^{n-1}$ and $|\chi_{x_i x_j}| = 2^{n-2}$, e.g. there are 2^2 rows of the above truth table where $x_2 = 1$ and 2^1 rows where $x_2 = x_1 = 1$. This limits our ability to perform partition refinement with signatures.

In this example, it is clear that x_2 cannot be moved to x_1 or x_0, since doing so would change f_1. Let us see if the available signatures indicate this fact. As the tables in Figure 3.6 show, the signatures $|\chi_{y_1 x_0}|$, $|\chi_{y_1 x_1}|$, and $|\chi_{y_1 x_2}|$ are identical. As we just explained, other signatures are identical by construction. Thus, to show that x_2 cannot be moved to x_1, we need to explore the permutations which move x_2 to x_1 and confirm that they are not symmetries.

3.5.2 Minterm Comparison

Minterm-based techniques [PR94, HK98, CS03] compare the value of a function at various points of Boolean space. For example, given a single-output function $f(x_2, x_1, x_0)$, we compare the values of $f(1,0,0)$, $f(0,1,0)$ and $f(0,0,1)$ at the root of a search tree. Assuming their values are 1, 0 and 0 respectively, we can conclude that moving x_2 to x_1 or x_0 would change f. This results in the variable partition $\{\{x_2\}, \{x_1, x_0\}\}$. We systematically examine f for other values of x to refine the partition or confirm that x_0 can be swapped in x_1.

The main drawback of this approach is that we may have to examine f for 2^n values of x. Our work in [CK06] reduces this number by applying the same transformations used in this chapter.

Two alternative extensions exist in order to find symmetries of a multiple-output function:

1. Convert to a single-output characteristic function. Converting f to a characteristic function χ eliminates many of the properties which [CK06] relies on. For example, given $f_0 = \bar{x}_0$ the characteristic function $\chi(x_0, y_0) = x_0 \oplus y_0$ is binate.
2. Add explicit support for multiple-output functions. This requires extensive bookkeeping to record which variables are unate for which outputs, etc. This facility is already provided to us in a graph automorphism solver, as we have shown in this chapter.

Thus, minterm-based techniques are not suitable for finding symmetries of multiple-output functions.

3.6 Previous Work for Other Types of Symmetry

For completeness, we note that other algorithms for finding symmetry have been proposed, for alternative definitions of symmetry. If we say that x_0 and x_1 can be swapped without changing f, this is equivalent to the follow equality:

$$f_{x_0\bar{x}_1} = f_{\bar{x}_0 x_1}$$

The authors of [EH78, CJ99, Mis03, KK08] consider other equalities such as

$$f_{x_0 x_1} = f_{\bar{x}_0 x_1}$$

generalized symmetries. These symmetries are "generalized" in the sense that they compare arbitrary pairs of cofactors, and do not necessarily correspond to permutations or negations of variables. This definition is less general than ours in that each symmetry involves only two variables.

The authors of [YM91, Mau06] explore symmetries which entail not only negation and/or permutation of inputs and outputs, but also the addition of XORs at the inputs. The concept is proposed as an additional source of flexibility in circuit implementation, but no complete algorithm to find the symmetries is provided.

3.7 Conclusion

The search for functional symmetry forms a cornerstone of our work, since in order to *use* symmetries, we must first find them. A number of prior works have proposed algorithms to solve the symmetry finding problem, but do not perform well when repurposed for multiple-output functions.

Rather than extend these specialized algorithms to work well for multiple-output functions, we reduce the symmetry finding problem to graph automorphism, since generic solvers which implement backtrack search with partition refinement are readily available. The partition refinement procedure is generally application-specific and critical to the performance of the solver because it helps prevent fruitless pursuit of symmetry. We incorporate the refinement rules presented in [CK06] to reduce the sizes of the graphs and reduce the time required in backtrack search.

We presented our graph formulation, and showed that it results in small graphs and exposes more information to the refinement rule Equation 3.2. Experimental results confirm that our formulations are solved efficiently by the Saucy graph automorphism package.

4. Symmetries in Circuits

In the previous chapter, we detected the symmetries of a function by reducing it to a graph for analysis by graph automorphism software. The procedure operates on a function's implicants, which may be too numerous. In addition, the procedure does not provide any insight into how to analyze and optimize a circuit. This chapter serves to transition into the rest of this dissertation by addressing the following concerns:

- Can we quickly find a function's symmetries without enumerating its implicants?
- How should we find the symmetries within a circuit?
- How will we use symmetries in circuits?

This chapter is organized as follows. We first review previous work for finding symmetries of circuits based on netlist automorphisms, and discuss conditions for when these algorithms find *all* symmetries of the corresponding function—a netlist must be a tree of associative and "prime" functions. When a netlist satisfies these conditions, the optimization potential of symmetries is also plainly laid out. We discuss how these symmetries can be categorized into three types, which will then be used in the next three chapters:

1. symmetries of individual components, in technology mapping
2. circuit symmetries, in placement
3. symmetries at primary inputs, in Boolean satisfiability

Finally, since real netlists typically do not come in the form of trees, we discuss heuristics for extracting portions which implement "prime" functions.

4.1 Boolean Networks

A **netlist** is a tuple of components and connections (V, n, m, E), where V is the set of components and $E \subseteq V \times \mathbb{N} \times V \times \mathbb{N}$ is the set of connections. Functions $n\colon V \to \mathbb{N}$ and $m\colon V \to \mathbb{N}$ record the number of **input** and **output ports** of each component, i.e. component $v \in V$ has $n(v)$ input ports. The input ports of each component v are numbered from 1 through $n(v)$, and the output ports are numbered from 1 through $m(v)$. Connections are made between ports, and the tuple (u, i, v, j) is a valid connection from u to v only if $1 \leq i \leq m(u)$ and $1 \leq j \leq n(v)$. Furthermore, exactly one connection is made to any input port, that is for any $v \in V$ and $j \in \{1, \ldots, n(v)\}$ the set $\{(u, i) : (u, i, v, j) \in E\}$ has a cardinality of 1. A **primary input** is a component with no inputs ($n(v) = 0$) and a **primary output** is a component with no outputs ($m(v) = 0$).

Some authors conflate the terms "graph" and "netlist", which we use to describe different things. We stress that a graph is similar to but *not* a netlist, no more than a set is a sequence of elements—a netlist allows multiple connections between two components, and the ports of a component are distinct. Clearly, we can discard information from a netlist to obtain a graph $\mathfrak{G} = (V, \{(u, v) : (u, i, v, j) \in E)$. Each primary input would then be called a **source** in $\mathfrak{G}$, and each primary output a **sink** in $\mathfrak{G}$.

A **Boolean network** is a netlist where every component v is associated with a Boolean function $f(v)\colon \mathbb{B}^{n(v)} \to \mathbb{B}^{m(v)}$. Note the similarity to the notation used in the previous chapter, which manipulates some $f\colon \mathbb{B}^n \to \mathbb{B}^m$. We adopt the convention used in previous literature and call each component in a Boolean network a "node". A node which is not a primary input nor output is called an **internal node**.

Thus, the motivation for distinguishing "graph" from "netlist" is clear—the inputs to each Boolean function $f(v)$ are not generally interchangeable.

4.2 Structural Symmetries of Boolean Networks

Suppose that we would like to find the symmetries of a function f. The previous chapter provides a method to formulate this function as a graph of implicants to be analyzed for automorphisms. An alternative approach is to first express f as a graph $\mathfrak{N}$, where each node implements a function which is symmetric with respect to its local set of inputs, e.g. AND and INV functions. We color each node according to its function, e.g. all nodes implementing AND are assigned color 0, and nodes implementing INV are assigned color 1. It is clear that any automorphism of $\mathfrak{N}$ corresponds to a symmetry of f, i.e. $Aut(\mathfrak{N}) \subseteq Sym(f)$. However, many symmetries of f may not exist as automorphisms of $\mathfrak{N}$, which motivates an extension to find more symmetries.

The authors of [WKSV03] propose two extensions to this concept to create a graph $\mathfrak{N}'$ which exposes more symmetries of f as automorphisms. If $\mathfrak{N}$ is an AND/INV graph, then $Aut(\mathfrak{N}) \subseteq Aut(\mathfrak{N}') \subseteq Sym(f)$. They specify the following rules for constructing $\mathfrak{N}'$:

1. Each node implements the AND function, INV function, or is a primary input.

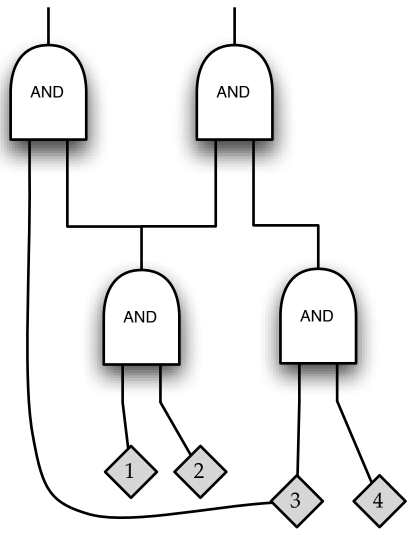
2. Each INV node has an outdegree of 1, and a pair of INV nodes in succession is replaced by a direct connection.

3. Each primary input is represented by a pair of strongly connected nodes denoting an input variable and its negation. No INV nodes are connected to primary inputs—AND nodes connect to one of the nodes in an input pair as necessary.

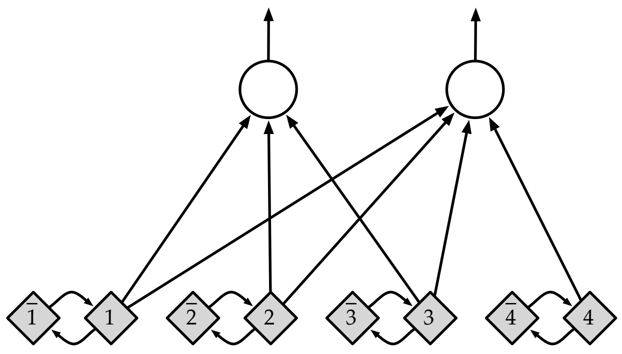
4. For a given AND node, any AND node at its inputs is absorbed. For example, a subgraph corresponding to AND($X \cup \{$AND(Y)$\}$) is replaced by AND($X \cup Y$).

The combination of the above rules implies that $\mathfrak{N}'$ consists entirely of NOR functions, since AND nodes are separated by an odd number of INV nodes—this construction is called a "NOR graph" in [WKSV03]. By making each local function depend on a maximal number of inputs, f is guaranteed to be expressed uniquely when $\mathfrak{N}'$ is a tree. The proof of a generalization of this assertion appears in [Ash59], and we will revisit this concept in the next section.

Let us now relax our notation so that x_i is denoted by i, and $\bar{x}_i$ is denoted by $\bar{i}$. Furthermore, the motion of negative literals is omitted from descriptions of permutations when they follow their positive counterparts, i.e. we write $(1,2,3)$ instead of $(1,2,3)(\bar{1},\bar{2},\bar{3})$.



(a) Boolean network $\mathfrak{N}$



(b) Graph $\mathfrak{N}'$

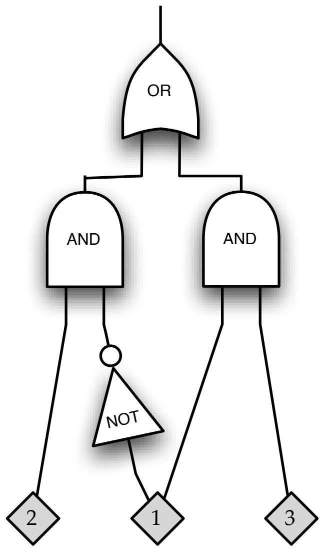
Figure 4.1: Boolean network and corresponding "NOR graph"

Example 4.1. Figure 4.1a shows a Boolean network which computes the two functions $x_3x_1x_2$ and $x_1x_2x_3x_4$. The graph in Figure 4.1b computes the same two functions, but exposes the symmetry $(1,3)$ as an automorphism.

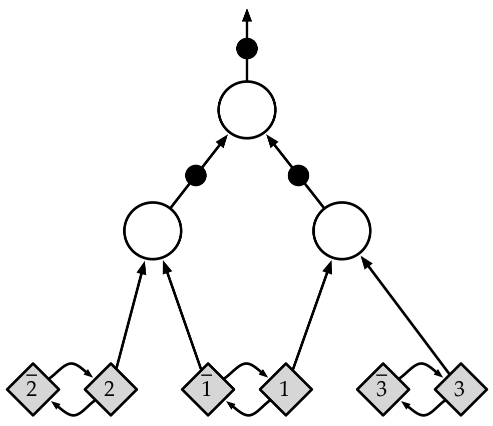
Example 4.2. Figure 4.2a shows a Boolean network which computes the MUX function $\bar{x}_1x_2 + x_1x_3$. This network has no automorphisms. By representing each primary input as a pair of nodes, as in Figure 4.2b, we find the symmetry $(1,\bar{1})(2,3)$. The symmetry $(2,\bar{2})(3,\bar{3})(y,\bar{y})$ is neither present in $\mathfrak{N}$ nor in $\mathfrak{N}'$.

Note that if $\mathfrak{N}$ is *not* an AND/INV graph, $Aut(\mathfrak{N}) \subseteq Aut(\mathfrak{N}')$ does not necessarily hold. Suppose that $\mathfrak{N}$ consists of a single node implementing the function $ab + ac + bc$. All symmetries of this function are automorphisms of $\mathfrak{N}$. If we use the distributive law in constructing (first an AND/INV graph and then) $\mathfrak{N}'$, we may obtain a graph which is isomorphic to $ab + c(a + b)$. The symmetry (a, b) is present in $\mathfrak{N}'$, but (a, c) is not. In order to expose every symmetry of f, we need to encode f in a canonical manner, such as the formulation presented in the previous chapter. The next section combines these concepts to define a canonical Boolean network for any function, so that all symmetries can be found.

To conclude this discussion, we repeat that the method proposed in [WKSV03] finds many symmetries of a function, but not all. The original motivation behind this



(a) Boolean network $\mathfrak{N}$



(b) Graph $\mathfrak{N}'$

Figure 4.2: Boolean network and corresponding "NOR graph"

approach was to detect symmetries quickly. In contrast, for our purposes we will use $\mathfrak{N}'$ in later chapters to classify symmetries which are not useful in synthesis.

Remark. Certain graph automorphism software packages, e.g. Saucy [DLSM04], assume that the graph to be analyzed is undirected. If the graph is directed and acyclic, then we assign each node a color such that nodes which have different depths (or levels) are colored differently. The depth (or level) of a node is the length of the longest path from a source (or primary input) to that node. A strongly connected pair of nodes is considered a single node for the purpose of computing levels. Having recolored the graph, we can proceed to compute automorphisms assuming that edges are undirected.

4.3 Structural Symmetries of Maximal Decompositions

Definition. A function f defined over a set of variables X is said to possess a **disjoint decomposition** if it can be written in the form $f(X) = g(X_0, h_1(X_1), h_2(X_2), \ldots, h_k(X_k))$, where the X_i form a partition of X. It is assumed that the decomposition is nontrivial, that is $k > 0$ and $|X_i| > 1$ for $i > 0$. If f does not have a nontrivial decomposition, we say that it is **prime**. We say that f is **maximally decomposed** if:

- f is one of the associative functions (AND, XOR) with inputs or output possibly negated, or
- f is prime, otherwise
- f is expressed as $g(X_0, h_1(X_1), h_2(X_2), \ldots)$ and the functions g and h_i are themselves maximally decomposed.

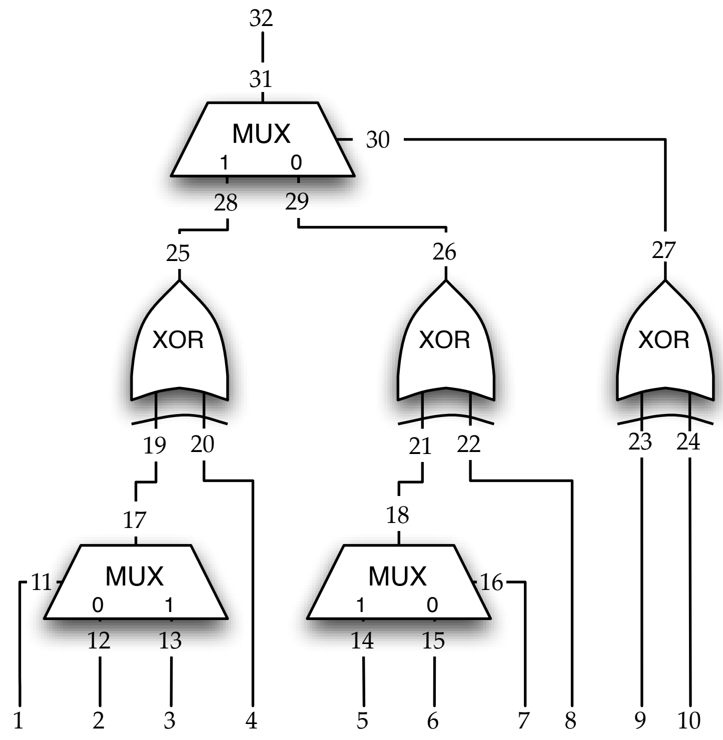
The first case has precedence over the third case—AND(a,b,c) can be decomposed into AND(AND(a,b),c), but the maximal decomposition is defined to be AND(a,b,c). Thus the term "maximal" implies both that decomposition is performed recursively, and that associative functions in the decomposition have a maximal number of inputs.

Theorem 2 (Ashenhurst, 1956 [Ash59])**.** *Suppose that f is maximally decomposed. This decomposition is unique modulo negation and permutation of the inputs and output of each local function.*

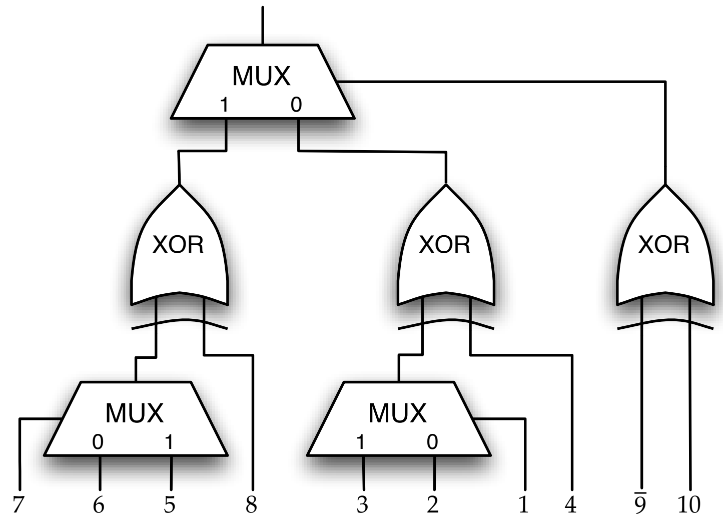
Example 4.3. Suppose that we are given the circuit in Figure 4.3a. Each component in the diagram represents a prime or associative function, respectively MUX or XOR. Thus, this circuit represents a maximal decomposition of the corresponding function f. A different maximal decomposition is given in Figure 4.3b, which is equivalent modulo the permutation/negation $(1,7)(2,6)(3,5)(4,8)(9,\bar{9})$—negating one of the inputs of the rightmost XOR negates the "select" input of the root MUX, which then exchanges its "0" and "1" inputs.

We have defined equivalence and symmetry for Boolean functions, and isomorphism and automorphism for graphs. Boolean networks are graph-like, but generally are not graphs because the function of a component may not be symmetric. Therefore, let us now define automorphism over Boolean networks. For comparison, let us first describe in an abstract manner the automorphism computation of graph $\mathfrak{G} = (V, E)$ as a computation over groups:

1. Group G specifies that for any $v \in V$, all edges to v are interchangeable.
2. Group B specifies that nodes of the same degree are interchangeable.



(a)



(b)

Figure 4.3: Two "isomorphic" maximal decompositions. The second decomposition swaps inputs {1,2,3,4} with {7,6,5,8} while negating 9.

3. Group C specifies that edges are preserved, i.e. Equation 3.3 holds.

For any groups G, B, the union $G \cup B$ is defined to be the smallest group that contains both G and B. Then the automorphisms of $\mathfrak{G}$ are described by the group $(G \cup B) \cap C$. The automorphisms of a decomposition are found with a similar sequence of computations:

1. We introduce a variable for each connection point to an internal node in the decomposition. More precisely, we introduce two points i and $\bar{i}$ to Ω representing a variable and its negation.

2. For each internal component b, we find its symmetries $Sym(b)$ and a permutation π_b leading to its canonical representation.

3. We define an initial group consisting of the symmetries of each internal component:

$$G = \bigcup_b Sym(b)$$

4. Suppose internal components b and c have the same number of input variables. Let $\pi_{b,c}$ denote the permutation which swaps the inputs of b with those of c and simultaneously swaps their outputs. Any automorphism of the decomposition only swaps equivalent components. Therefore, let

$$B = \langle \{ \pi_b \pi_{b,c} \pi_c^{-1} : b \text{ and } c \text{ have the same canonical representation} \} \rangle$$

5. We define a group which allows primary inputs and outputs to move freely:

$$\begin{aligned} I = \langle &\{\text{all permutations and negations of primary inputs}\} \\ &\cup \{\text{all permutations and negations of primary outputs}\} \rangle \end{aligned}$$

   We separate I from G and B for reasons which will be clear later.

6. We constrain each variable pair corresponding to the ends of a connection to move in unison. One end of a connection may be negated iff the other is as well.

$$\begin{aligned} C = \langle &\{ (i, u)(j, v) : i \text{ connects to } j, \text{ and } u \text{ connects to } v \} \\ &\cup \{ (i, \bar{i})(j, \bar{j}) : i \text{ connects to } j \} \rangle \end{aligned}$$

Using the groups we have just defined, we present two corollaries which follow directly from Ashenhurst's theorem:

Definition. Let X be the symmetric group over all variables representing internal nodes. The subset of permutations in X which do not change the overall function of the Boolean network are called **circuit symmetries**.

Corollary 1. *The circuit symmetries of a maximal decomposition are described by $G \cup B$. In other words, unlike the example in Figure 1.4, the circuit symmetries of a maximal decomposition form a group.*

Definition. The group $(G \cup B \cup I) \cap C$ forms the automorphisms of a decomposition.

Corollary 2. *Every symmetry of f corresponds to an automorphism of its maximal decomposition, and $|Sym(f)| = |(G \cup B \cup I) \cap C|$.*

Example 4.4. Let us compute the automorphisms of the circuit in Figure 4.3a. The groups G, B, I and C are defined below:

$$
\begin{aligned}
G = \langle\{ & \\
& \left.\begin{array}{l}(12,\overline{12})(13,\overline{13})(17,\overline{17}) \\ (12,13)(11,\overline{11})\end{array}\right\}\text{symmetries of M}\textsc{ux}\text{ component} \\
& \qquad\vdots \\
& \left.\begin{array}{l}(19,20) \\ (19,\overline{19})(25,\overline{25})\end{array}\right\}\text{symmetries of X}\textsc{or}\text{ component} \\
& \qquad\vdots \\
\}\rangle & \\
B = \langle\{ & \\
& \left.\begin{array}{l}(11,16)(12,15)(13,14)(17,18) \\ (11,30)(12,29)(13,28)(17,31)\end{array}\right\}\text{can exchange M}\textsc{ux}\text{ components} \\
& \left.\begin{array}{l}(19,21)(20,22)(25,26) \\ (19,23)(20,24)(25,27)\end{array}\right\}\text{can exchange X}\textsc{or}\text{ components} \\
\}\rangle & \\
I = \langle\{ & \\
& \left.\begin{array}{l}(1,2) \\ (1,2,3,4,5,6,7,8,9,10)\end{array}\right\}\text{can exchange primary inputs} \\
& (1,\overline{1})\ \}\text{ can negate primary inputs} \\
& (32,\overline{32})\ \}\text{ can negate primary output} \\
\}\rangle & \\
C = \langle\{ & \\
& \left.\begin{array}{l}(1,11)(2,12) \\ (1,11)(3,13) \\ \qquad\vdots \\ (1,11)(27,30) \\ (1,11)(31,32)\end{array}\right\}\text{can exchange connections} \\
& (1,\overline{1})(11,\overline{11})\ \}\text{ connection variables are negated in unison}
\end{aligned}
$$

The final result $(G \cup B \cup I) \cap C$ has the following set of generators:

$$\begin{aligned}(G \cup B \cup I) \cap C = \langle\{&(9,\bar{9})(10,\overline{10})(23,\overline{23})(24,\overline{24}),\\ &(9,10)(23,24),\\ &(1,\bar{1})(2,3)(11,\overline{11})(12,13),\\ &(2,\bar{2})(3,\bar{3})(4,\bar{4})(12,\overline{12})\cdots(20,\overline{20}),\\ &(1,7)(2,6)(3,5)(4,8)(9,\bar{9})(11,16)\cdots(28,29)(30,\overline{30})\}\rangle\end{aligned}$$

By construction, each permutation moves an entire subtree of the decomposition. This explains why $|Sym(f)| = |(G \cup B \cup I) \cap C|$. We can then project these generators onto permutations over the primary inputs and outputs only:

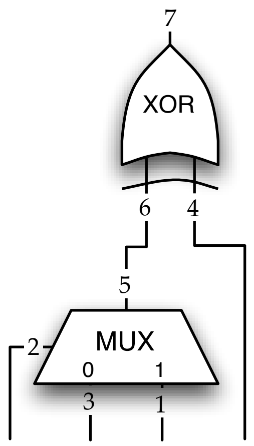
$$\begin{aligned}Sym(f) = \langle\{&(9,\bar{9})(10,\overline{10}),\\ &(9,10),\\ &(1,\bar{1})(2,3),\\ &(2,\bar{2})(3,\bar{3})(4,\bar{4}),\\ &(1,7)(2,6)(3,5)(4,8)(9,\bar{9})\}\rangle\end{aligned}$$

Thus, this leads to a method to find *all* of the symmetries of a function, provided that we are given (or can obtain) a maximal decomposition—we again turn to regular graph automorphism to perform the necessary computation.

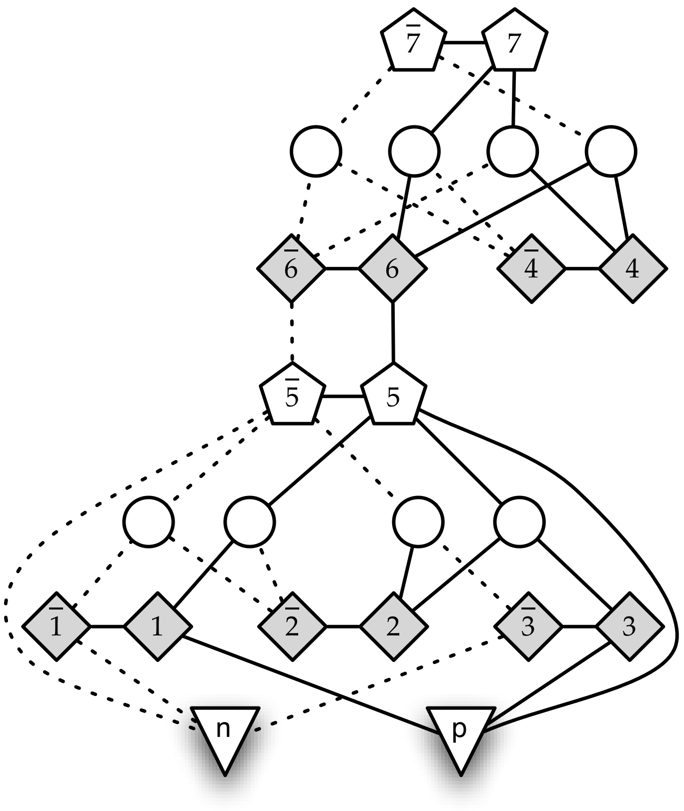
4.3.1 Reduction to Graph Automorphism

We have just shown how to find the automorphisms of a decomposition by finding the symmetries for each component in the decomposition and combining them with a series of group calculations. For each connection in the decomposition, we introduced a pair of variables as input and output. Each variable is then represented by a pair of points, for the variable and its negation.

Having placed ordinary graph automorphism into the same framework, it is straightforward to reduce Boolean network automorphism into ordinary graph automorphism. For each component in a Boolean network, we create a "widget" using the formulation presented in the previous chapter. Each edge in the Boolean network translates directly into a pair of edges between widgets. Figure 4.4a shows a Boolean network, and Figure 4.4b shows its conversion into an ordinary graph.



(a) Decomposition



(b) Graph automorphism encoding

Figure 4.4: Graph automorphism encoding of a decomposed function

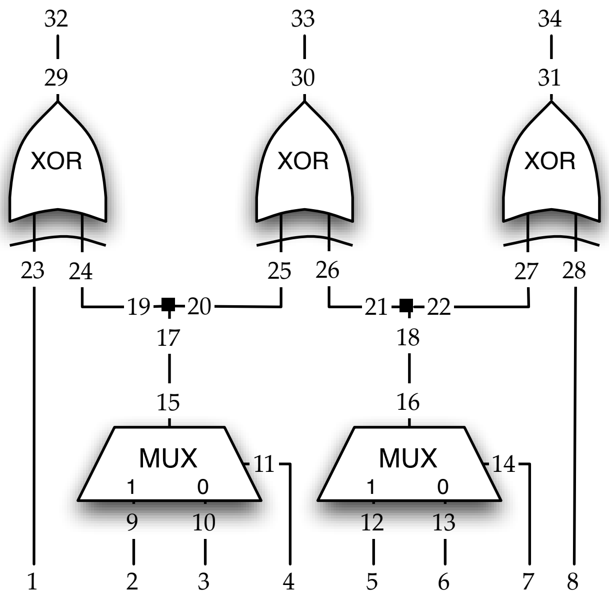


Figure 4.5: Maximal decomposition in the form of a polytree

4.3.2 Polytrees

Definition. A **polytree** is a directed acyclic graph which has at most one path between any pair of nodes. We will say that a netlist is a polytree iff its underlying graph is a polytree. A netlist which is not a polytree is said to have **reconvergence**, and a pair of nodes which have multiple paths between them are said to have **reconverging paths**.

The discussion so far has centered on single-output functions, but most circuits have multiple outputs. The same results hold for multiple-output functions, if we assume a decomposition in the form of a polytree. Every multiple-output function has a polytree decomposition; in the worst case we obtain a tree decomposition for each output and merely share primary input nodes.

The group formulation for a polytree is similar to that for a tree. We explicitly add a junction point to the Boolean network for each node with multiple fanouts. Each junction point can be thought of as a component with one input and multiple outputs which each compute the identity function. This simplifies the construction by keeping all connections point-to-point. Groups G, B, and C are defined in the same way as before. Group I is augmented with permutations which swap junctions of the same degree.

Example 4.5. For the polytree in Figure 4.5, we define I as follows:

$$\begin{aligned}
I = \langle\{ & \\
& \left.\begin{array}{l}(1,2)\\(1,2,3,4,5,6,7,8)\end{array}\right\}\text{can exchange primary inputs}\\
& \left.(1,\bar{1})\right\}\text{can negate primary inputs}\\
& \left.\begin{array}{l}(32,33)\\(32,34)\end{array}\right\}\text{can exchange primary outputs}\\
& \left.(32,\overline{32})\right\}\text{can negate primary outputs}\\
& \left.\begin{array}{l}(19,20)\\(21,22)\end{array}\right\}\text{outputs of junctions are interchangeable}\\
& \left.\begin{array}{l}(17,\overline{17})(19,\overline{19})(20,\overline{20})\\(18,\overline{18})(21,\overline{21})(22,\overline{22})\end{array}\right\}\text{propagate negated values}\\
& \left.(19,21)(20,22)(17,18)\right\}\text{can exchange junctions}\\
\}\rangle &
\end{aligned}$$

The automorphisms of the polytree are again $(G \cup B \cup I) \cap C$, shown below. Cycles of internal points are not shown.

$$\begin{aligned}
(G \cup B \cup I) \cap C = \langle\{ & (2,3)(4,\bar{4})\cdots,\\
& (2,\bar{2})(3,\bar{3})\cdots(32,\overline{32})(33,\overline{33}),\\
& (8,\bar{8})\cdots(34,\overline{34}),\\
& (1,8)(2,5)(3,6)(4,7)\cdots(32,34)\}\rangle
\end{aligned}$$

4.4 Applications in CAD

This dissertation addresses the application of symmetries in CAD, and we have just defined three groups of interest for a given maximal decomposition:

- The symmetries of all components, G.
- Circuit symmetries, $G \cup B$.
- The automorphisms of the Boolean network, $(G \cup B \cup I) \cap C$.

Our motivation for using symmetries is to increase the design space (for synthesis), and to shrink the search space (for verification). For each application in this dissertation, one of these three groups will be useful for this goal, while the other two will not be. Thus, it is important to analyze each problem to use symmetries efficiently and correctly before formulating any algorithms to solve them. We tackle the problems of technology mapping, placement, and Boolean satisfiability. In this section, we continue with the assumption that the Boolean network with which we are presented is a maximal decomposition.

4.4.1 Technology Mapping

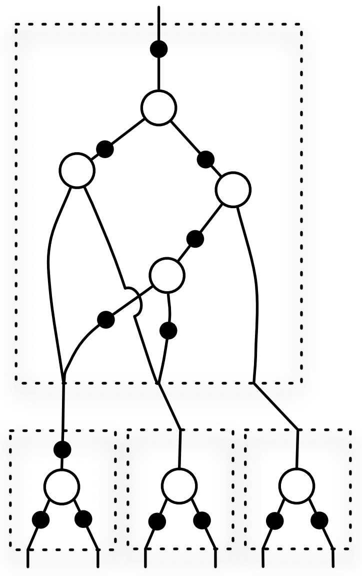
For the technology mapping problem, we are presented with a subject graph of AND and INV primitives and tasked to find a covering of this graph which minimizes some objective. We use symmetries to derive a *different* subject graph which implements the same overall function, because the best covering of this new graph may be better than the best covering of the original graph.

Suppose there exists a covering/partitioning of our subject graph which is isomorphic to the maximal decomposition. Then we can permute the inputs of each block according to G in order to create a new subject graph. Let us assume that blocks in the decomposition which implement equivalent functions contain identical subgraphs. Then exchanging the contents of two equivalent blocks will not modify the subject graph at all. Therefore we do not use group B for technology mapping. Similarly, we do not use group $(G \cup B \cup I) \cap C$.

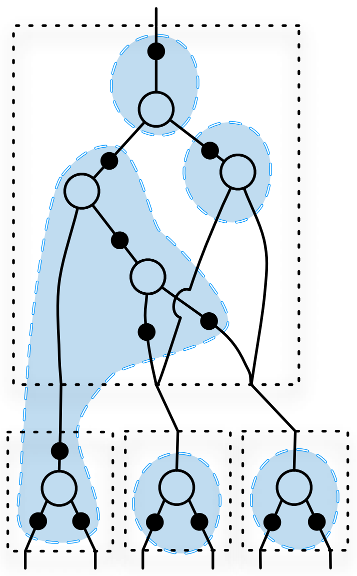
Example 4.6. Figure 4.6a shows a subject graph similar to that in Figure 5.5a, where white circles represent AND primitives and black circles represent INV primitives. By permuting the contents of the topmost block, we obtain a new graph (Figure 4.6b) which permits a different covering. Exchanging the contents of the two blocks on the lower right (as per group B) would not change the subject graph in a meaningful way.

4.4.2 Placement

In the placement problem, we would like to restructure a netlist in order to minimize some objective which is determined by the physical locations of the components. Therefore, exchanging connections between identical components under group B will affect our objective since identical components will have different locations under any legal placement. Furthermore, the ports on each component have distinct locations, and exchanging connections under group G may also affect our objective. Typically, the locations of primary



(a) Subject graph which forms a maximal decomposition



(b) Same subject graph, with inputs to top block permuted

Figure 4.6: Subject graphs. White circles are AND primitives, small black circles are INV primitives.

inputs and outputs are fixed and therefore group $(G \cup B \cup I) \cap C$ does not apply to the placement problem.

In summary, we optimize over group $G \cup B$ for the placement problem.

4.4.3 Boolean Satisfiability

The Boolean satisfiability problem searches for an assignment to $x \in \mathbb{B}^n$ such that $f(x)$ is true. Suppose we know some symmetry $\pi \in Sym(f)$, i.e. $f = f^\pi$. Then $f(x)$ is false iff $f(x^\pi)$ is false—there is no need to explicitly check the value of both. Thus, given a decomposition for f, we find the automorphisms $(G \cup B \cup I) \cap C$ which equate to the symmetries of f. We then add a **symmetry breaking predicate** ϕ which evaluates to true for a "transversal" of $Sym(f)$. More precisely, $Sym(f)$ creates equivalence classes of $\mathbb{B}^n$, and ϕ is used to select one element (or more) from each. The formula $f = (x_1 + x_2)$ exhibits the symmetry (x_1, x_2) which implies that $f(0,1) \equiv f(1,0)$. We can exclude the case $x_1 = 1, x_2 = 0$ by forming the constraint $\phi = (\bar{x}_1 + x_2)$, and directing a SAT solver to solve $f\phi$.

Symmetries of a *subformula* of f cannot be used for symmetry breaking predicates unless if they are also symmetries of f. Suppose $f = gh$ where $g = (x_1 + x_2)$ and $h = (x_1)(\bar{x}_2)$. Subformula g exhibits the symmetry (x_1, x_2) which can be broken with the predicate ϕ from above. Subformula h does not exhibit the same symmetry. The unsatisfiability of $gh\phi$ does not imply that gh is also unsatisfiable—in fact, $gh\phi$ is unsatisfiable while gh can be satisfied with $x_1 = 1, x_2 = 0$.

In conclusion, the groups G and B represent symmetries of subformulas and do not themselves lead to valid symmetry breaking predicates for SAT problems. We must use the automorphisms $(G \cup B \cup I) \cap C$.

4.5 Decomposition Heuristic

So far in this chapter, we have analyzed the symmetries of maximally decomposed functions and discussed the use of these symmetries in three application contexts. In the case of SAT, it is sufficient to find the symmetries of a formula with no regard to the formula's internal structure, since symmetries of *subformulas* are not used. For technology mapping and placement, the structure of the decomposition will dictate the symmetries available to us. When the decomposition forms a (poly)tree, we find and use the groups G and B for technology mapping and placement, knowing that these groups fully describe the optimization potential over permutations and negations of connections within the decomposition.

However, subject graphs and standard cell netlists are not presented in the form of a polytree decomposition.

First, subject graphs and standard cell netlists are composed of AND/INV primitives or standard cells, respectively, rather than prime functions. It is not obvious how to recover a maximal decomposition when the prime functions are further decomposed into smaller units. Figure 4.6b shows the same function decomposed three different ways: AND/INV primitives (white and black circles), standard cells (blue or shaded "blobs"), and prime functions (dotted rectangles). The graph of AND/INV primitives represents the function at a very low level—it would be easier to derive the unique (modulo permutation) decomposition and the AND/INV graph which follows, than to work backwards from an unstructured AND/INV graph.

Second, a subject graph or netlist may not be decomposed maximally. One reason is that signals in the maximal decomposition may not be present in the netlist. For example, the OAI22 cell in Figure 4.6b spans two decomposition blocks, and connections within a standard cell may not be modified. Another reason is possible sharing between outputs. Suppose we start with a high-level description f of a circuit which specifies that the two outputs implement x_1x_2 and $x_1x_2x_3$, respectively. The maximal decomposition of f simply consists of two nodes, $\textsc{And}(x_1, x_2)$ and $\textsc{And}(x_1, x_2, x_3)$. A direct translation to an AND/INV graph yields:

$$\begin{aligned} y_1 &= \textsc{And}(x_1, x_2) \\ t &= \textsc{And}(x_1, x_2) \\ y_2 &= \textsc{And}(t, x_3) \end{aligned}$$

The variables $\{x_1, x_2, x_3\}$ can be arbitrarily permuted among the last two AND nodes. In contrast, the smallest AND/INV graph consists of two AND nodes:

$$\begin{aligned} y_1 &= \textsc{And}(x_1, x_2) \\ y_2 &= \textsc{And}(y_1, x_3) \end{aligned}$$

The first AND is shared between the two outputs, meaning that connections to x_1 and x_3 cannot be exchanged without changing the first output. This limits the amount of flexibility at our disposal.

Thus, a subject graph or standard cell netlist may not be a maximal decomposition, and if it were, it would be difficult to find the boundaries of each block in the decomposition. Instead, we use a heuristic guided by the principles of maximal decomposition to select regions of a subject graph (or standard cell netlist).

To simplify the discussion, let us assume the task of finding symmetries for use in technology mapping. As before, we would like to isolate regions of the subject graph to perform modifications over. Each region defines a local function over the inputs to the region, and we find the symmetries with respect to those local inputs, which we denoted earlier by G. A large number of subgraphs can be selected from a graph, and it would be infeasible to explicitly search for symmetries on each subgraph. For some subgraphs, we may not find any symmetries. For other subgraphs, we may find so-called "higher-order" symmetries (symmetries in $G \cup B$) which signify wasted effort. This motivates a heuristic to find regions which represent prime functions. Regions representing ANDs of maximal size are handled specially because they can be found in linear time.

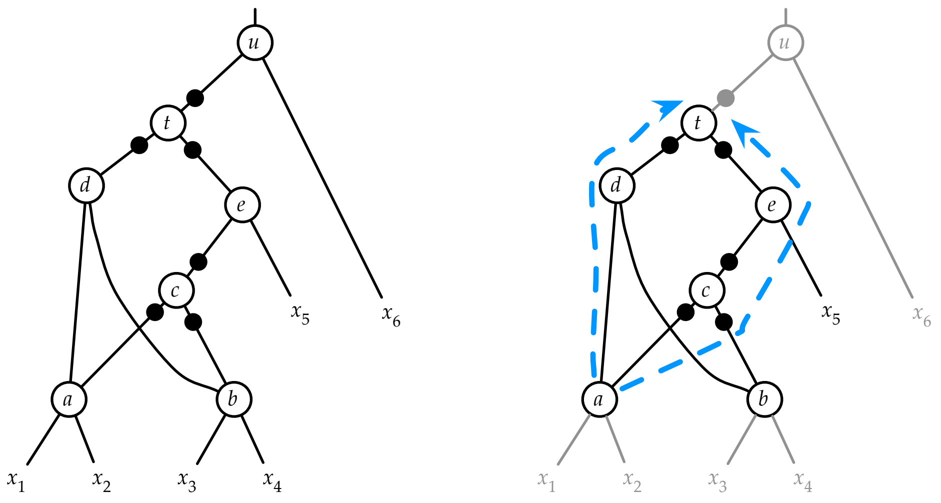
Definition. In a directed graph, a node d **dominates** node u if all paths from u to the sinks passes through d. Node d is the **immediate dominator** of u if no other node dominates all paths from u to d.

Theorem 3. *Let $f(x)$ be the function described by an* AND/INV *graph $\mathfrak{G}$. An output f_j is prime* only if *there exists an input x_i such that there are at least two distinct paths from source x_i to sink y_j, where the two paths share no common prefix or suffix besides the endpoints.*

Proof. For f_j to be prime, $\mathfrak{G}$ must contain some reconvergence. Otherwise f_j is represented by a tree of associative functions (AND nodes) with inputs or outputs possibly negated (INV nodes) and is therefore not prime.

More generally, suppose there exists a partition of the variables X into sets X_0, X_1, $X_2, \ldots$ such that (a) there exists only one path from each $x_i \in X_0$ to y_j and (b) all $x_i \in X_k$ for $k > 0$ are dominated by t_k where $t_k \neq y_j$. Then f_j can be decomposed into some function $g(X_0, t_1, t_2, \ldots)$ and is therefore not prime. $\square$

By the above theorem, any subgraph we select should contain reconverging paths



(a) Graph (b) Subgraph with inputs a, b, x_3

Figure 4.7: A graph which contains a subgraph (rooted at t) implementing a prime function

from an input which culminate at an output node. This places a restriction on the nodes which can act as outputs of subgraphs, as well as a minimum limit on the size of each subgraph—a subgraph must be large enough to have a source of reconvergence as an input.

Example 4.7. Let us analyze the AND/INV graph in Figure 4.7a. Any subgraph with u as its output is not prime because all nodes/variables on the left side are dominated by t, and there is only one path from x_6 to u. The subgraph with t as its output and $\{a, x_3, x_4, x_5\}$ as inputs *may* be prime because t is the immediate dominator of a.

The issue at this stage is the *maximum* size of a subgraph. For efficiency, we restrict subgraph inputs to multifanout nodes dominated by output nodes, and "side inputs" which must be included to define a proper subgraph. This also serves as a heuristic to maximize the amount of symmetry found.

Example 4.8. Let us assume a subgraph rooted at t in Figure 4.7b. Nodes x_1 and x_2 are dominated by a, therefore we select node a to serve as an input, which has two paths to t: $a \to d \to t$ and $a \to c \to e \to t$. Node b is an input to c and d and must be included in the subgraph, either as an internal node or as an input node. Similarly, node x_5 is a side input to e and must be included in the subgraph. We thus extract the subgraph with output t and inputs a, b, x_5.

We incorporate these heuristics into Algorithm 6. For a node u, $out(u)$ denotes $\{v : (u, v) \in E\}$; similarly $in(v) = \{u : (u, v) \in E\}$. The immediate dominator of node u is denoted by $dom(u)$. It is assumed that nodes are topologically sorted, i.e. $u < v$ for all $v \in out(u)$. Furthermore, AND nodes are expanded maximally up to multifanout nodes—a tree of ANDs is replaced by a single AND of all the tree's inputs.

The algorithm proceeds in three passes to find a subgraph with r as a sink. The first (backwards) pass finds a set of nodes I such that each $u \in I$ has one path of length l to r, has at least one other path to r, and is not dominated by some t_k where $t_k < r$. The second (forward) pass marks the paths from nodes in I to r. The third pass defines inputs X and outputs Y which form a proper subgraph.

4.6 Conclusion

In this chapter we reviewed the concept of "structural symmetries" as presented in [WKSV03]. The idea is to find the symmetries of a function by obtaining the automorphisms of a netlist. It follows from Ashenhurst's theory of decomposition [Ash59] that these two sets are equal if the netlist forms a maximal decomposition composed of associative functions and prime functions. The equality of these two sets allows us to reason precisely about the potential use of symmetries in optimizing such netlists, and we showed how three groups of symmetries are used in three different applications.

Practical circuits generally do not form maximal decompositions, and therefore we have proposed a heuristic to extract subcircuits which implement prime-like functions. This heuristic uses the fact that AND/INV graphs of prime functions must have reconvergence.

The next three chapters discuss efficient algorithms to use our newfound symmetries in technology mapping, placement, and Boolean satisfiability.

Algorithm 6 EXTRACT-PRIME(r, l)

```
for each $u \in V$ do
    $\delta[u] \leftarrow \infty$
end for
queue $q$ is initialized to $r$
$\delta[r] \leftarrow 0$
$I \leftarrow \emptyset$
while $q$ is not empty do
    $v \leftarrow$ remove front element of $q$
    if $\delta[v] < l$ then
        for each $u \in in(v)$ do
            if $\delta[u] = \infty$ then
                $\delta[u] \leftarrow \delta[v] + 1$
                append $u$ to back of $q$
            else if $dom(u) \geq r$ and $\delta[u] = l$ then
                $I \leftarrow I \cup \{u\}$
            end if
        end for
    end if
end while

$q \leftarrow I$
while $q$ is not empty do
    $u \leftarrow$ remove front element of $q$
    if $u \notin S$ then
        $S \leftarrow S \cup \{v\}$
        for each $v \in out(u)$ such that $\delta[v] \neq \infty$ do
            append $v$ to back of $q$
        end for
    end if
end while

$X \leftarrow \emptyset$
$Y \leftarrow \emptyset$
for each $v \in S$ do
    if $\{u : u \in in(v) \wedge u \in S\} = \emptyset$ then
        $X \leftarrow X \cup \{v\}$
    else
        $X \leftarrow X \cup \{u : u \in in(v) \wedge u \notin S\}$
    end if
    if $\exists w \in out(v)$ where $w \notin S$ then
        $Y \leftarrow Y \cup \{v\}$
    end if
end for
```

5. Technology Mapping

In this chapter, we use circuit symmetries of subject graphs in order to improve the results of technology mapping. Technology independent logic synthesis algorithms typically aim to minimize the size of a Boolean network in terms of literals or of an AND/INV graph in terms of AND nodes: this is used as a heuristic to minimize the size of the final implementation whether as in the number of FPGA look-up tables or area consumed by standard cells.

The size of an AND/INV graph is not a perfect indicator of the size of the final implementation, and two AND/INV graphs of equal size may lead to implementations of different sizes. We propose two algorithms which permute nodes in a subject graph to aid in technology mapping. This chapter is organized as follows. First, we review the basic technology mapping procedure which uses cuts, matching, and covering. Then we review previous work in adding choices to subject graphs, and our algorithms to exploit symmetry. Finally, we report the results of our algorithms after technology mapping, and conclude the chapter.

5.1 Introduction

Technology mapping is invoked after the general logic synthesis phase. Given a **subject graph** of primitives (generally 2-input AND gates and inverters), a technology mapper obtains a logically equivalent network consisting of components from a technology library.

Usually these are standard cells, but may be lookup tables or combinational logic blocks in the case of FPGAs.

Technology mapping consists of two phases:

1. enumerating subgraphs and **matching** them against library cells, followed by
2. performing a **covering** of the graph with the matches found.

The matching step defines a solution space for the following covering step. A covering is valid if each primary output is produced by a component, and for each component, its inputs are in turn produced by other components or by primary inputs.

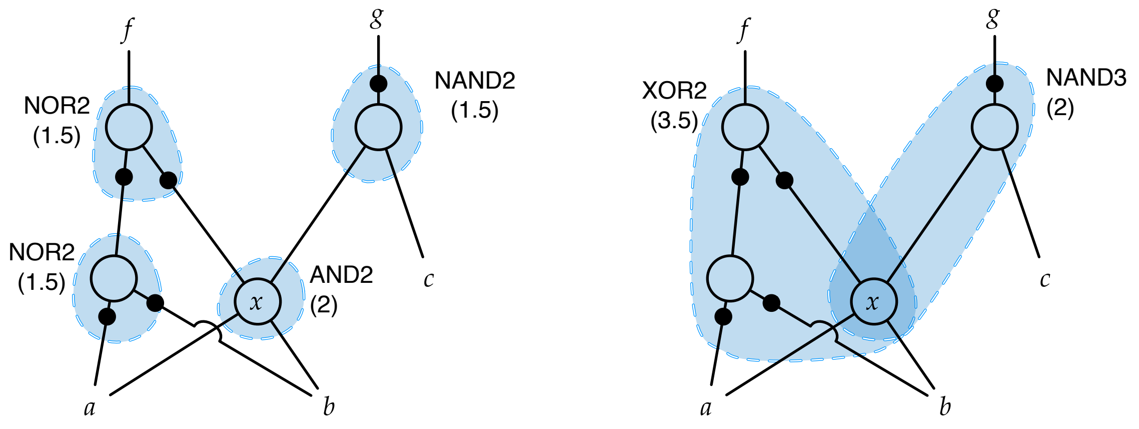
The first general mapping procedure (DAGON) was described in [Keu87]. Previously, each new technology library would require a new mapping procedure. The DAGON procedure partitions a circuit into leaf-dags,[1] and for each leaf-dag:

1. enumerates matches using structural patterns of AND and INV primitives, then
2. finds a minimum-cost matching using a dynamic programming algorithm.

The solutions for the leaf-dags are then stitched together to form the solution for the entire subject graph. While the general approach is sound, the DAGON procedure is limited by the fact that it only considers leaf-dags and that it limits itself to the structure dictated by the provided subject graph. The first limitation has two effects: first, it removes the possibility of matching against components which have multiple outputs, and second, it disallows the covering of a subject graph node by multiple components. Figure 5.1 illustrates the case where covering subject graph node x twice leads to a smaller final circuit: in one particular technology, the circuit in Figure 5.1a has an area of 6.5, while the one in Figure 5.1b has an area of 5.5.

The second limitation of the DAGON procedure, that it is limited to the structure dictated by the subject graph, is illustrated in Figure 5.2. The matching step of the mapping procedure implies that every signal in the final result must have existed in the subject graph. The upper circuit in Figure 5.2 may be more desirable, but the gate AND(c,b) cannot result from the subject graph because there is no subgraph implementing the function cb. Thus, it is important to have a "good" subject graph beforehand in order to produce a good final circuit.

[1] A tree in which leaves are permitted to appear multiple times.



(a) Without duplication (b) With duplication

Figure 5.1: Duplication of a subject graph node leads to a smaller circuit (areas in parentheses from Nangate library)

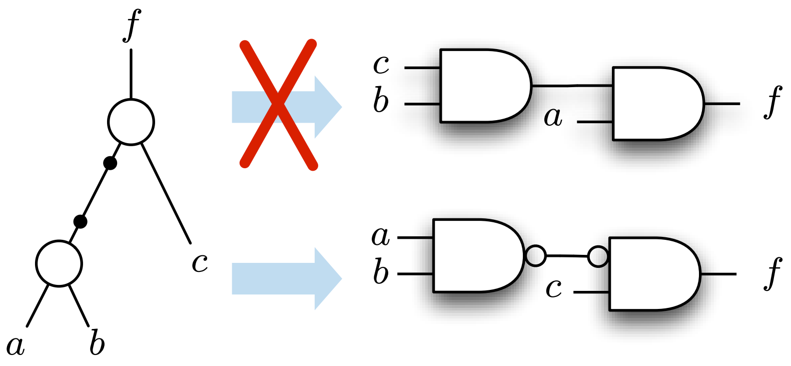


Figure 5.2: Some circuits cannot be obtained without modifying the subject graph.

The objective of logic synthesis then, is to obtain a good subject graph for the technology mapper. Logic synthesis usually attempts to obtain a small (in terms of the number of AND primitives) and/or shallow subject graph to minimize the final area and/or delay of the circuit. However, due to the following reasons, the size of the subject graph does not necessarily correlate to the size of the final circuit:

- Inverters in the subject graph are assigned a cost of 0, but usually have nonzero cost in a technology library.
- Nodes with multiple fanouts may need to be covered multiple times, i.e. duplicated, to obtain a smaller circuit.
- Different technology libraries are constructed using different design rules, and favor certain gates more than others.

To illustrate the relative strengths of different technology libraries, Table 5.1 shows the cell areas of three freely available cell libraries, normalized to the area of an inverter. We focus on cell area in our discussion because it is easier to quantify than other metric such as delay. A few trends are apparent:

- The GSC library has different costs for cells from AND and OR families (NAND4 versus NOR4 and AOI21 versus OAI21)
- The GSC library favors MUX and XOR gates.
- The Nangate library favors AOI and OAI gates.

Example 5.1. The function $f(a, b, c)$ can be expressed as $a(b\bar{c} + \bar{b}c) + \bar{a}bc$, or as $(ab + c(a + b))(\overline{abc})$. Both expressions consist of 7 AND/INV nodes, shown in Figure 5.3. However, the circuit in Figure 5.3a is preferable if using the GSC library, and the circuit in Figure 5.3b is preferable if using the Nangate library. The logic synthesis step which precedes technology mapping has no way to determine which respective subject graph should be generated.

Cell	Function	Normalized Size		
		GSCLib [Cad05]	Nangate [Nan08]	vsclib [Pet08]
INV	$\bar{a}$	1	1	1
NAND2	$\overline{ab}$	1.25	1.5	1.3
NAND3	$\overline{abc}$	1.75	2	1.7
NAND4	$\overline{abcd}$	2	2.5	2.3
NOR2	$\overline{a+b}$	1.25	1.5	1.3
NOR3	$\overline{a+b+c}$	3	2	1.7
NOR4	$\overline{a+b+c+d}$	4.75	2.5	2.3
AND2	ab	1.5	2	1.7
OR2	$a+b$	1.5	2	1.7
AOI21	$\overline{a_1a_2+b}$	1.5	2	1.7
AOI22	$\overline{a_1a_2+b_1b_2}$	2	2.5	2.3
AOI211	$\overline{a_1a_2+b+c}$	—	2.5	2
OAI21	$\overline{(a_1+a_2)b}$	2	2	1.7
OAI22	$\overline{(a_1+a_2)(b_1+b_2)}$	3	2.5	2.3
OAI211	$\overline{(a_1+a_2)bc}$	—	2.5	2
MUX2	$\bar{s}a+sb$	3	4	2.7
XOR2	$\bar{a}b+a\bar{b}$	2.75	3.5	2.7
XNOR2	$\bar{a}\bar{b}+ab$	—	5	2.7

Table 5.1: Normalized sizes for standard cells from three libraries. When multiple sizes (e.g. NAND2X1 or NAND2X2) exist, the smallest is shown. A dash ("—") denotes that no such cell exists in the library.

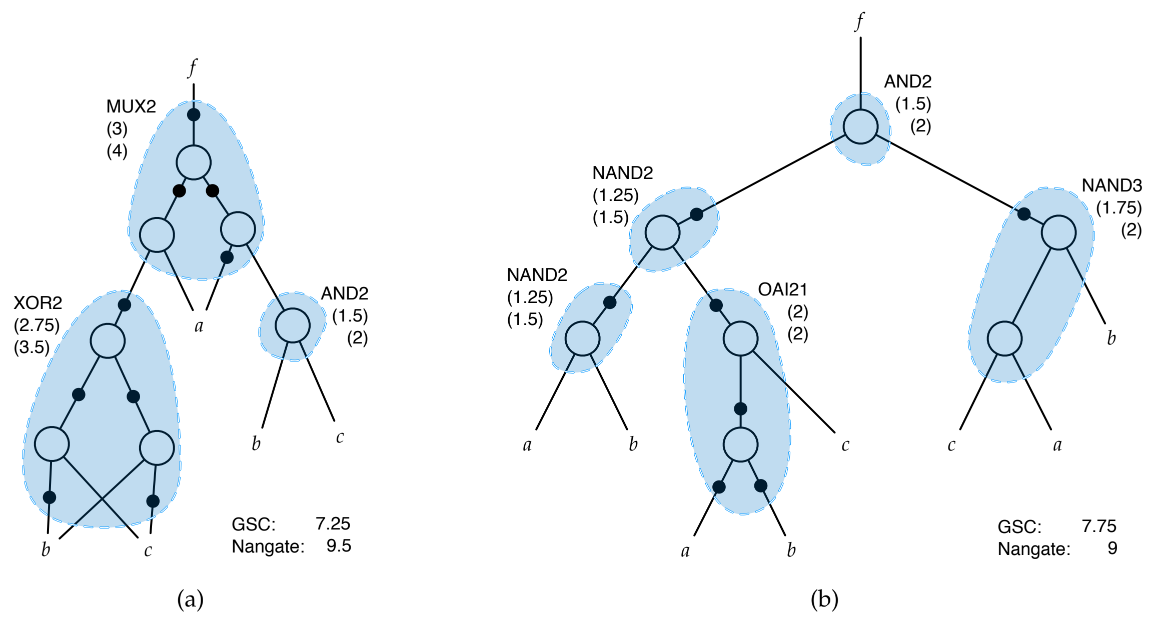


Figure 5.3: Two implementations of the same function. Cell areas with GSC and Nangate libraries shown in parentheses.

5.2 Previous Improvements in Technology Mapping

The problem we face in technology mapping is that the mapping procedure is too highly dependent on the structure of the subject graph. A number of previous works have sought to address some of the deficiencies in the DAGON procedure. We have grouped them according to the problem they address.

5.2.1 Cuts, Boolean Matching, and DAG covering

The first problem in DAGON we cited was the reliance on leaf-dags to perform matching and covering. An alternative procedure based on cuts and Boolean matching removes these restrictions, and solves the mapping problem on a more abstract level.

A ***k*-feasible cut**[k@*k*-feasible cut] [CD94, CWD99] for a node u in a subject graph is a set of k or fewer nodes X such that every path from the primary inputs to u passes through a node in X. Sets of cuts are defined recursively:

- For a primary input u, $\textsc{k-Cuts}(u) = \{\{u\}\}$.

- For a node u = $\textsc{And}(a, b)$,

$$\textsc{k-Cuts}(u) = \{\{u\}\} \cup \{c_1 \cup c_2 : c_1 \in \textsc{k-Cuts}(a), c_2 \in \textsc{k-Cuts}(b), |c_1 \cup c_2| \le k\}$$

- For a node u = $\textsc{Inv}(a)$, $\textsc{k-Cuts}(u) = \textsc{k-Cuts}(a)$.

Each k-feasible cut defines a subgraph with sources X and sink u, and a function $u = f(X)$.

We then identify a component from our target technology which implements f, if any. For FPGAs which use lookup tables (LUTs), $f(X)$ can be implemented by a LUT iff $|X|$ is lesser than or equal to the number of inputs to a LUT. For standard cell libraries, the task of **Boolean matching** [MM90] identifies a component g and permutation/negation π such that $g^\pi = f$.

Finally, we select a valid covering of the subject graph. This is called a "binate" covering problem [Rud89, VKBSV97] because of the presence of constraints for a feasible covering—a component may be used iff the values at its inputs are provided by other components.

5.2.2 Choices and Supergates

The second problem in DAGON, that every component in the final implementation must have a corresponding node u in the subject graph, is ameliorated through the use of **choice nodes** [LWGH97]. Choice nodes are used to encode multiple logically equivalent subject graphs into a single structure. The work in [LWGH97] enumerates choices based on algebraic rules such as distributivity: $a(b + c) = ab + ac$. However, the concept of choices is not limited to algebraic rewriting. An ordinary AND/INV graph can be obtained from an enriched graph by replacing every edge from a choice node by an edge from either of the choice node's inputs.

For example, we can combine the two subject graphs in Figure 5.3 with a choice node that says "choose Figure 5.3a or Figure 5.3b". We also add a choice node indicating that $(bc)a$ and $b(ca)$ are equivalent; the result shown in Figure 5.4 represents three different AND/INV graphs. The technology mapper is free to select any of the choices that improves its objective. The difference between embedding choices into the subject graph and choosing the best of n subject graphs is that the effect of choices is multiplicative, so one would have to use many choice-less subject graphs. (The variant proposed in [LWGH97] allows

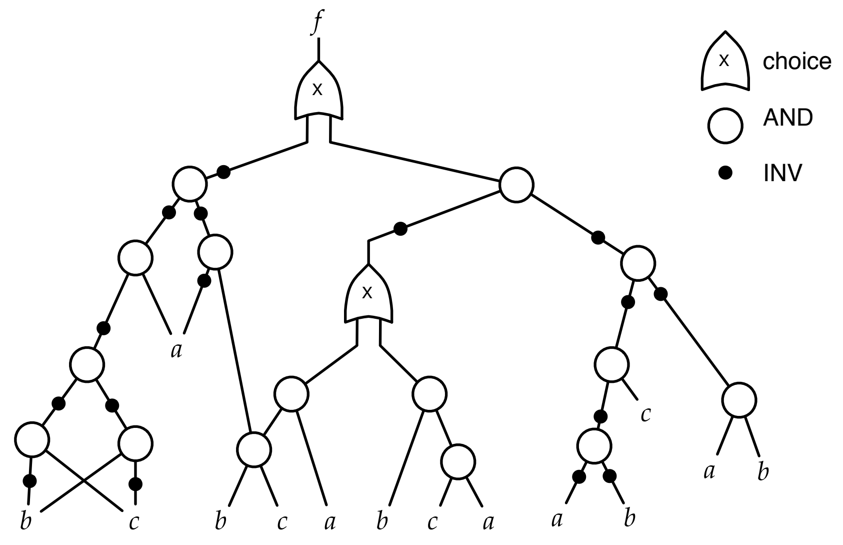


Figure 5.4: Subject graph with choices formed by combining Figure 5.3a and Figure 5.3b and adding a choice for associativity

cycles to occur in the enriched subject graph, which amplifies the effect of choices even further.) Another difference is a matter of efficiency: some choices may be eliminated early on in the technology mapper.

However, the addition of choices carries with it a cost as the technology mapper must explore a larger search space. Furthermore, since many technology mapping algorithms are based on heuristics, increasing the design space may worsen the final quality.[2] Then, we need a more global mechanism to filter out poor choices in the subject graph, rather than rely on the somewhat more local mechanism in the technology mapper itself.

Another limitation of the procedure described in [LWGH97] is that *only* algebraic manipulations are performed: the Boolean axioms are not used. On the other hand, by restricting themselves to algebraic manipulations, the authors may claim to exhaustively enumerate algebraic choices.

The work published in [CMB$^+$05b] overcomes the structural dependency by simply merging several logically equivalent subject graphs. Given a subject graph, they generate a new graph using several *local* Boolean operations. Then for a set of subject graphs, they identify internal points which are equivalent; each set of equivalences is now a set of choices in the final enriched graph.

[2]Many previous works place an emphasis on reducing the number of cuts enumerated [CWD99, MCB07]

Another method proposed in [CMB+05b] combines multiple technology components into a **supergate**. A supergate is a macro component consisting of several individual technology components; the benefit of using a supergate is that the internal point(s) of a supergate need not exist in the subject graph. For example, we can create two different supergates: one supergate based on the covering in Figure 5.3a and one supergate based on Figure 5.3b. In this case, we do not need choice nodes to encode the respective subject graphs, because the complexity is shifted to the Boolean matcher. For any subject graph representing $f = ab\bar{c} + a\bar{b}c + \bar{a}bc$, we can use the cut $X = \{a, b, c\}$ and query the Boolean matcher for a component (here a supergate) g and permutation/negation π such that $g^{\pi} = f$. The disadvantage of supergates is that we need to explicitly enumerate and add them to our technology library.

A complete technology mapping procedure which incorporates cut enumeration, Boolean matching, choices, and supergates is described in [Cha07].

5.3 Creating Choices Through Symmetry

Rather than create choices using unpredictable and ad-hoc logic synthesis steps, we propose to use symmetries to make changes to a subject graph. Since symmetries by definition only permute and/or negate connections, they do not increase the number of AND nodes in the subject graph.

Example 5.2. The function $f = ab + c(a + b)$ is symmetric in all of its variables. Therefore another valid representation is $f = cb + a(c + b)$. This function occurs in the subject graphs in Figure 5.5, where $a = d + e$. As the figure shows, the two representations lead to different minimal implementations.

As a general mechanism, we can use symmetries to generate choices in a subject graph. Given any subgraph of a subject graph, we find the functional symmetries and add the respective images of the subgraph as choices. For example, given the subject graph in Figure 5.5a, we permute the connections from $X = \{a, b, c\}$, obtaining its image in Figure 5.5b. We then add this image as a choice.

Following the theory laid out in the previous chapter, we use the symmetries in group G, which are the symmetries of AND/XOR trees and of prime functions. For various reasons, these two types of functions are processed differently:

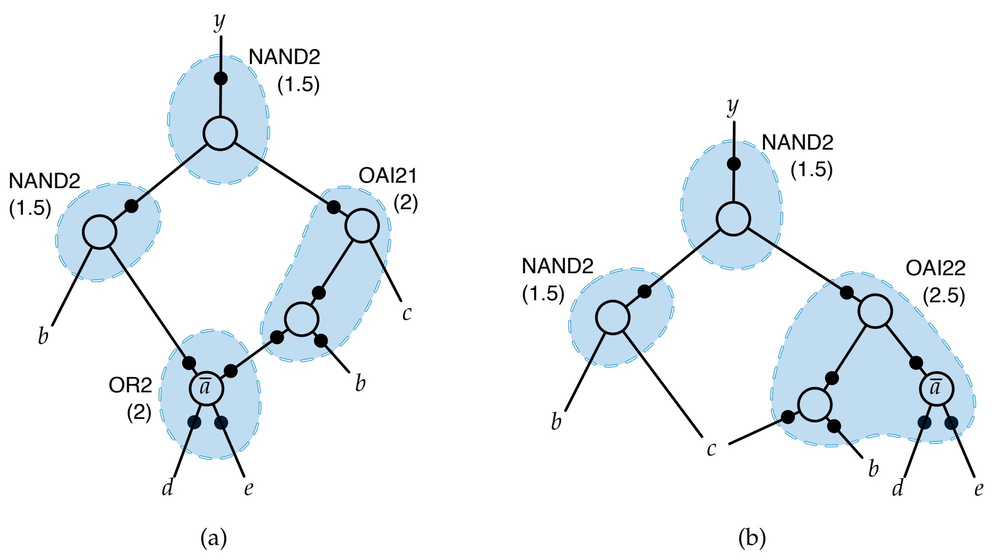


Figure 5.5: Permuting subgraph nodes according to symmetry creates an additional mapping option.

1. AND and XOR trees are easier to identify than prime functions.

2. Not only may the inputs of AND/XOR trees be arbitrarily permuted, these trees can also be restructured in any way.

3. For efficiency, we need to limit the number of AND/XOR restructurings generated, since any n-input AND function can be represented by $\mathcal{O}(2^n)$ different trees of 2-input ANDs.

4. Many of the symmetries of a prime function f are also symmetries of f's constituent AND trees in the subject graph. Example 5.4 will illustrate this point.

5.3.1 AND/XORS

The associative and commutative properties of the AND (and XOR) functions are typically used to restructure trees. For example, we can minimize circuit depth by balancing paths in a Huffman-like manner: beginning with a set of variables x_i and arrival times for each x_i, we can construct a minimum depth AND tree by iteratively removing two variables with lowest arrival time and replacing them with the AND of the two [Cor03].

With this procedure, we can restructure every AND tree in a circuit in topological order, and minimize the total depth of the circuit. However, unless our cell library consists purely of 2-input AND gates and inverters, this does not necessarily minimize the delay of the mapped circuit, simply because we don't use the actual arrival times during the balancing procedure. In addition, this does not help the case where we want to minimize the total cell area consumed by the final implementation.

We propose an alternative procedure which examines the overall structure of the subject graph in order to perform modifications. This procedure is based on the concept of k-feasible cuts introduced previously. As we described, a k-feasible cut X for a node corresponds to a possible implementation of the node using some $f(X)$ from the technology library. For two nodes x_0 and x_i with respective cuts X and X', the AND of x_0 and x_i has a cut $X \cup X'$. If X and X' overlap, it is possible that $X \cup X'$ is k-feasible. Thus, our heuristic procedure tries to reorder the leaves of a tree of ANDs in order to increase overlaps between cuts.

Recall that each node has a (potentially large) set of k-feasible cuts. Rather than explicitly compute the overlaps between the cuts for each pair of nodes, we maintain a count of the number of occurrences of each node in each node's set of cuts. Then for each pair of nodes, we compare their counts—if there is a high overlap in the counts, it is highly likely that one cut from each of their cutsets will overlap. The following example illustrates this principle.

Example 5.3. Figure 5.6a shows an AND-tree which implements $x_1(x_2(x_3x_4))$. At the next-to-lowest level, the internal node y has no small cut in terms of nodes $\{a, b, \ldots, h\}$—the 4-feasible cuts are $\{\{x_4, a, b, d\}, \{x_4, a, e, f\}, \{x_3, f, g, h\}\}$. For each input x_i to the AND-tree, we calculate a vector representing the number of occurrences of other nodes in its k-input cuts. For node x_1, $c^1 = (1\,1\,1\,0\,0\,0\,0\,0)^{\mathrm{T}}$, and for x_3, $c^3 = (2\,1\,0\,1\,1\,1\,0\,0)^{\mathrm{T}}$. We compute a weight $w_{1,3} = c^1 \cdot c^3 = 3$. Weights for other pairs are defined similarly, while $w_{i,i}$ is left undefined.

$$W = \begin{pmatrix} \cdot & 0 & 3 & 0 \\ 0 & \cdot & 1 & 3 \\ 3 & 1 & \cdot & 1 \\ 0 & 3 & 1 & \cdot \end{pmatrix}$$

Algorithm 7 CUT-BALANCE(X)

```
1: if $|X| \leq 2$ then
2:     return AND($X$)
3: else
4:     for each $x_i \in X$ do
5:         $C \leftarrow k$-input cuts for $x_i$
6:         for each $u \in X$ do
7:             $c_u^i \leftarrow$ number of occurrences of node $u$ in $C$
8:         end for
9:     end for
10:    for each $x_i, x_j \in X$ do
11:        $w_{i,j} \leftarrow \sum_u c_u^i c_u^j$
12:    end for
13:    select a partition $X_1, X_2$ which minimizes $\sum_{x_i \in X_1} \sum_{x_j \in X_2} w_{i,j}$
14:    return AND(CUT-BALANCE($X_1$), CUT-BALANCE($X_2$))
15: end if
```

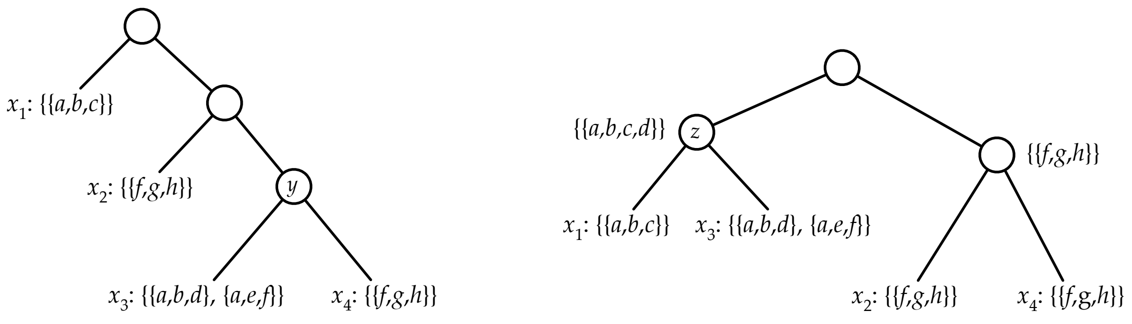


Figure 5.6: Rewriting AND-tree to merge cuts

Then we partition the inputs $\{x_1, x_2, x_3, x_4\}$ into sets X_1 and X_2 to minimize

$$\sum_{x_i \in X_1} \sum_{x_j \in X_2} w_{i,j}$$

For this example, we choose $X_1 = \{x_1, x_3\}$ and $X_2 = \{x_2, x_4\}$, which results in an objective value of $w_{3,2} + w_{3,4} = 2$. We recursively partition these two sets, and finally obtain the tree AND(AND(x_1, x_3), AND(x_2, x_4)) as shown in Figure 5.6b. As we can see, node z has a small cut $\{a, b, c, d\}$ because its children x_1 and x_3 have overlapping cuts.

Note that the parameter k for the k-feasible cuts (line 5 of Algorithm 7) is not necessarily the same one used for the actual technology mapping step, and that certain values of k may lead to better results.

5.3.2 Prime Functions

The other type of function in a maximal decomposition is a prime function—a function which does not have a disjoint decomposition. In the previous chapter, we described a heuristic algorithm EXTRACT-PRIME which targets regions containing reconvergence because they are likely to represent prime functions.

For each region, we compute the symmetry group G for its corresponding function f. We may then add the image of this region for any permutation $\pi \in G$ as a choice representing an alternative implementation of f, since $f^{\pi} = f$. For two reasons, we do not proceed exactly in this manner:

1. Since EXTRACT-PRIME does not exactly partition a circuit into prime functions, we need to run EXTRACT-PRIME with various values of the depth parameter l.

2. Since the subject graph is composed of AND and INV primitives, many of the symmetries in G can also be thought of as symmetries of associative functions, and examined using the algorithm from section 4.2. The next example illustrates this concept.

We incorporate these two observations into the algorithm ADD-CHOICES.

Algorithm 8 ADD-CHOICES

```
1:  for each $u \in V$ do
2:      for each $l \in \{2, 3, 4, \dots\}$ do
3:          $X, Y \leftarrow$ EXTRACT-PRIME$(u, l)$
4:          $\mathfrak{N} \leftarrow$ the subgraph of $\mathfrak{G}$ bordered by
5:          $\mathfrak{N}' \leftarrow$ the NOR-graph of $\mathfrak{N}$ as described in section 4.2
6:          $f \leftarrow$ the Boolean function from $X$ to $Y$
7:          $G \leftarrow Sym(f)$
8:          $H \leftarrow Aut(\mathfrak{N}')$
9:          for each $\pi \in (G : H)$ do
10:             add $\mathfrak{N}^{\pi}$ to $\mathfrak{G}$ as a choice
11:         end for
12:     end for
13: end for
```

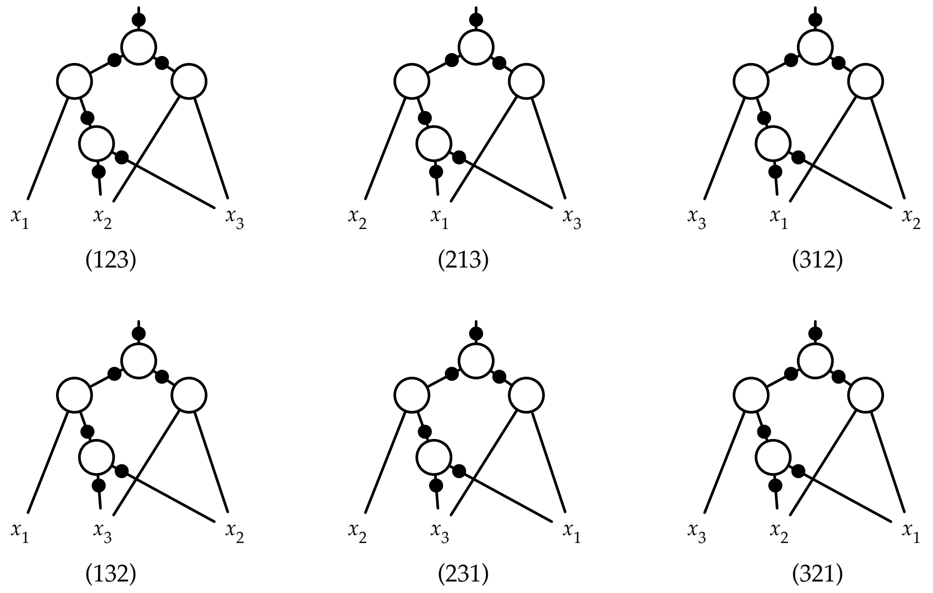


Figure 5.7: Image of an AND/INV graph for each permutation in S_3

Example 5.4. The function $f = x_2x_3 + x_1(x_2 + x_3)$ is symmetric in all its variables ($G = S_3$), and its AND/INV graph possesses a "structural symmetry" $H = \{(), (2, 3)\}$. Note that these are the same groups shown in Example 2.6. Figure 5.7 shows the image of the AND/INV (sub)graph for each permutation in S_3. Clearly, each graph in the top row is equivalent to the one immediately below it; the two corresponding permutations are a right coset of H in G. Meanwhile, the graphs in the top row are not isomorphic to one another, and the corresponding permutations form a right transversal $G : H$.

5.4 Experimental Results

We tested our improvements to synthesis on instances from the IWLS 2005 suite of benchmarks [IWL05]. We first converted each design to a combinational AND/INV graph by replacing each register with an input/output pair and each standard cell with its equivalent AND/INV nodes. We implemented our algorithms in ABC version 70930 [Ber], a state-of-the-art logic synthesis tool which includes facilities for technology mapping.

5.4.1 AND Trees

The first set of experiments tests the cut-based restructuring of AND functions described in Algorithm 7. We implemented the restructuring as an ABC command `fbalance`, which accepts a parameter k indicating the size of the cuts to be computed, and modifies every AND tree in a Boolean network using CUT-BALANCE. For comparison, the ABC command `balance` restructures AND functions to minimize circuit depth.

We first show the effects of restructuring on FPGA mapping. As a baseline, we apply the following sequence of ABC commands from [MBC08] on each benchmark:

```
resyn; resyn2           # rewrite netlist using algebraic rules
if -a -K 4              # map netlist onto K-input LUTs
choice                  # unmap netlist, and add choices with algebraic rules
if -a -K 4              # remap netlist onto K-input LUTs
choice; if -a -K 4      # repeat 3 more times
choice; if -a -K 4
choice; if -a -K 4
```

The script performs technology-independent optimizations on a netlist, and repeatedly maps the netlist onto a minimal number of 4-input LUTs. Between each pair of mapping iterations, we enrich the netlist using the built-in `choice` command. The `choice` command adds choices to a netlist based on the approach described in [CMB+05b]. The commands `resyn`, `resyn2` and `choice` all use the `balance` command through the alias `b`. To test the effect of CUT-BALANCE, we replace every use of `balance` with `fbalance`:

```
alias b 'fbalance -k 5'
resyn; resyn2
if -a -K 4
choice; if -a -K 4
choice; if -a -K 4
choice; if -a -K 4
choice; if -a -K 4
```

The results are presented in Table 5.2, comparing area (in LUTs) at each step of the two parallel scripts. Each pair of columns indicates the LUT count after a mapping step in the original script, and the relative *improvement* after the same step in the revised script. The following annotated script describes the results for `wb_conmax`.

```
load wb_conmax.aig
resyn; resyn2
if -a -K 4              # circuit now consists of 16357 LUTs
```

Benchmark	LUTs	%	LUTs	%	LUTs	%	LUTs	%	LUTs	%
pci_conf_cyc_addr_dec	43	0.00	43	0.00	43	0.00	43	-2.33	43	-2.33
steppermotordrive	63	19.05	60	20.00	48	0.00	48	0.00	49	0.00
ss_pcm	123	-0.81	123	-0.81	123	-0.81	123	0.00	123	0.00
usb_phy	183	1.64	183	1.64	181	1.66	182	2.20	180	1.67
sasc	205	0.49	197	-3.55	197	-3.55	197	-3.05	197	-2.54
simple_spi	283	-1.77	281	-1.42	278	-2.16	279	-1.79	279	-2.15
pci_spoci_ctrl	364	4.95	338	3.85	328	4.57	316	2.85	306	0.33
i2c	364	1.65	354	2.54	346	2.02	344	2.62	346	2.02
systemcdes	1071	7.38	1071	7.38	1000	7.50	995	4.22	965	4.77
spi	1218	6.90	1221	9.66	1129	4.34	1108	3.34	1099	3.37
wb_dma	1375	0.80	1350	1.19	1329	-0.38	1334	0.45	1325	-0.08
des_area	1918	3.23	1973	5.63	1901	4.58	1943	7.72	1864	3.38
tv80	2689	5.32	2615	7.88	2494	5.09	2473	5.05	2453	4.77
systemcaes	2944	-1.53	2900	-2.48	2829	-0.32	2767	-2.57	2754	-0.18
mem_ctrl	3480	12.39	2994	5.58	2918	5.24	2886	5.27	2864	5.31
ac97_ctrl	3870	0.59	3922	2.68	3908	1.92	3890	2.06	3827	0.13
usb_funct	4841	8.72	4716	8.14	4519	4.74	4471	4.12	4422	3.71
pci_bridge32	6083	-1.04	6041	-0.91	5988	-1.40	5978	-1.51	5963	-1.64
aes_core	8494	2.73	8446	2.27	8336	2.16	8307	2.26	8280	2.63
wb_conmax	16357	-0.02	15482	-2.18	14846	2.18	14553	2.02	14457	2.22
ethernet	20123	-2.08	20033	-2.59	19965	-2.74	19924	-2.55	19878	-2.86
vga_lcd	30455	-2.49	29972	-1.99	29926	-2.37	29933	-2.82	29959	-2.72
des_perf	33320	7.51	32782	7.74	31638	7.82	30900	7.17	30491	7.00
TOTAL	139866	1.96	137097	1.72	134270	1.80	132994	1.48	132124	1.34

Table 5.2: Cut-based restructuring of AND trees ($k = 5$), mapped to 4-input LUTs

```
choice; if -a -K 4      # 15482 LUTs
choice; if -a -K 4      # 14846 LUTs
choice; if -a -K 4      # 14553 LUTs
choice; if -a -K 4      # 14457 LUTs
load wb_conmax.aig
alias b 'fbalance -k 5'
resyn; resyn2
if -a -K 4              # circuit consists of 16361 LUTs ≈ 16357 × 100.02%
choice; if -a -K 4      # 15819 LUTs ≈ 15482 × 102.18%
choice; if -a -K 4      # 14523 LUTs ≈ 14846 × 97.82%
choice; if -a -K 4      # 14259 LUTs ≈ 14553 × 97.98%
choice; if -a -K 4      # 14136 LUTs ≈ 14457 × 97.78%
```

The first pair of columns in Table 5.2 shows an overall improvement of 1.96% after the first mapping iteration, for a total area of 137130. This is roughly equal to the LUT count achieved by otherwise simply performing another mapping iteration: 137097 LUTs. In addition, AND tree restructuring leads to a smaller relative area improvement when we perform many iterations of FPGA mapping. For example, if we perform five iterations of mapping to 4-input LUTs, the overall improvement drops to 1.34%. These raise the question of whether our AND tree restructuring can be removed in favor of performing one more iteration of FPGA mapping; this question is explored in the next experiment.

Previously, we used the `fbalance` command implicitly in *every* call to `resyn`, `resyn2`, and `choice`. To more accurately compare against a single iteration of FPGA mapping, we use `fbalance` during *one* call to `resyn/resyn2` or `choice` (which calls `resyn/resyn2`) and the original `balance` during all others. The results for the benchmark suite as a whole are shown in Table 5.3. Each row represents one additional iteration of FPGA mapping; the column "LUTs" denotes the LUT count assuming no use of `fbalance` and are identical to the figures in the last row of Table 5.2. The next six columns present the improvement gained from applying `fbalance -k 5` at the first, second, third, etc. call to `resyn/resyn2` or `choice`. For example, the following script

```
resyn; resyn2
if -a -K 4
choice; if -a -K 4
choice; if -a -K 4
```

yields a baseline set of circuits requiring 134270 LUTs, while

```
resyn; resyn2
if -a -K 4
```

Mapping iterations	LUTs	% improvement						
1	139866	1.96						
2	137097	2.32	2.67					
3	134270	1.78	2.05	1.27				
4	132994	1.40	1.41	1.62	1.07			
5	132124	1.26	1.08	1.19	1.13	0.56		
6	131957	1.31	1.19	1.20	1.18	1.31	0.62	
7	131205	1.01	0.81	0.77	0.78	0.95	0.91	0.07

Table 5.3: Effects of one series of AND tree restructuring at various points in the technology mapping flow

```
choice; if -a -K 4
alias b 'fbalance -k 5'
choice
alias b 'balance'
if -a -K 4
```

yields a 1.27% improvement. From the results, we can draw two conclusions. First, if we are to limit the use of `fbalance` for computational efficiency, we should use it earlier in the technology mapping process. Second, the gains from `fbalance` do not equate to a constant number of additional iterations of FPGA mapping. To achieve a 1.96% improvement over the 139866 LUTs at mapping iteration 1 requires one additional mapping iteration assuming no usage of `fbalance`. To achieve a 1.40% improvement over the 132994 LUTs after iteration 4 requires at least three more iterations. Therefore we conclude that CUT-BALANCE as implemented in `fbalance` provides a significant improvement that cannot otherwise be obtained by repeatedly performing FPGA mapping.

The next set of experiments explores the effect of the parameter k on the efficacy of `fbalance`, and the effect of `fbalance` when mapping to different LUT sizes K. Table 5.4 and Table 5.5 show the final LUT counts when we vary the LUT size $K \in \{4,5,6\}$ and parameter $k \in \{2,3,\ldots,K,K+1,K+2\}$ in the following script:

```
alias b 'fbalance -k k'
resyn; resyn2
if -a -K K
choice; if -a -K K
choice; if -a -K K
choice; if -a -K K
choice; if -a -K K
```

The results show that `fbalance` yields more improvement as K increases. In addition, the data indicate that a value of $k = 4$ or $k = 5$ works well in most cases, though $k = 2$ works best for $K = 4$.

Finally, we examine the effect of `fbalance` when performing technology mapping with standard cells. In order to encourage the technology mapper to minimize area, we modified the data for the GSCLib library [Cad05] so that every gate has a delay of 1. We then run the below script on each design to arrive at a baseline set of areas.

```
resyn; resyn2
read_library gsclibUNIT.genlib
map
```

Then we use `fbalance` with a parameter $k \in \{2, 3, 4, 5, 6\}$ for comparison.

```
alias b 'fbalance -k k'
resyn; resyn2
read_library gsclibUNIT.genlib
map
```

The results are shown in Table 5.6. As Table 5.1 shows, the standard cells all have 4 or fewer inputs. This predicts that `fbalance` will be ineffective for $k = 5$ or $k = 6$. In fact, the results show that using `fbalance` deteriorates total area for $k \neq 2$.

Thus far, we have shown that Algorithm 7 provides improvements for both FPGA and standard cell mapping. Care should be taken to use the proper parameter k depending on the target platform.

5.4.2 Prime Functions

We now present results with Algorithm 8 (ADD-CHOICES). In contrast to Algorithm 7 (CUT-BALANCE), Algorithm 8 is geared specifically for standard cell mapping rather than for FPGA mapping: the permutation of subject graph nodes is meant to permit more standard cell matches to be made. We implemented Algorithm 8 as the command `transversal` in ABC. To control the size of the subject graph, we impose a limit of 8 on the number of elements from $G : H$ on line 9 of Algorithm 8. We also add a command `choicetrans` which combines the choices added from the built-in `choice` command with our `transversal` command.

We again perform standard cell mapping with our modified version of GSCLib:

```
resyn; resyn2
read_library gsclibUNIT.genlib
map
```

Benchmark	$K = 4$						$K = 5$						
	LUTs	% improvement					LUTs	% improvement					
		$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$		$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$	$k = 7$
pci_conf…	43	-2.33	-2.33	-2.33	-2.33	-2.33	43	0.00	0.00	0.00	0.00	0.00	0.00
stepper…	49	2.04	2.04	0.00	0.00	0.00	41	0.00	2.44	2.44	0.00	2.44	4.88
ss_pcm	123	0.00	-0.81	0.00	0.00	0.00	121	1.65	0.83	0.83	0.83	0.83	0.83
usb_phy	180	2.78	2.22	1.67	1.67	1.67	157	0.64	1.27	1.91	-0.64	0.00	-1.27
sasc	197	0.00	-3.05	-2.54	-2.54	-3.05	164	0.00	0.00	0.00	0.00	0.00	0.00
simple_spi	279	3.94	1.08	-1.08	-2.15	-1.43	234	0.43	0.43	4.70	5.13	3.85	3.85
pci_spoci_ctrl	306	1.96	3.92	3.59	0.33	-0.33	256	4.30	-0.39	5.86	3.91	2.73	7.03
i2c	346	1.16	1.73	2.89	2.02	3.76	274	-2.55	-1.82	-0.73	-0.36	-2.92	0.36
systemcdes	965	4.87	6.01	5.28	4.77	5.49	638	4.39	10.50	14.11	14.26	14.58	13.01
spi	1099	0.73	4.91	3.09	3.37	1.09	801	-3.00	0.00	-1.50	-3.00	0.00	-0.87
wb_dma	1325	0.30	0.30	-0.91	-0.08	0.38	1165	0.00	0.69	0.26	1.37	0.26	0.60
des_area	1864	-0.38	5.53	3.54	3.38	3.27	1374	-0.58	0.36	1.38	2.69	1.75	2.11
tv80	2453	2.94	4.77	4.28	4.77	3.30	2024	3.80	3.71	5.29	6.47	5.53	4.84
systemcaes	2754	1.16	0.00	-0.29	-0.18	-0.29	2218	2.25	1.94	1.40	2.93	1.44	1.13
mem_ctrl	2864	0.94	2.79	4.33	5.31	5.52	2451	-0.16	1.63	3.79	3.75	4.12	4.12
ac97_ctrl	3827	1.78	0.76	0.29	0.13	0.44	3269	0.43	0.03	0.61	1.16	1.50	1.25
usb_funct	4422	3.46	3.64	3.64	3.71	3.01	3653	4.11	2.35	2.55	2.24	4.68	5.37
pci_bridge32	5963	0.35	-1.83	-1.32	-1.64	-1.02	5408	0.28	0.11	0.24	0.06	-0.04	-1.02
aes_core	8280	2.51	3.43	2.67	2.63	2.00	6340	1.88	4.76	4.64	3.08	2.00	0.71
wb_conmax	14457	3.12	-2.08	-0.57	2.22	0.95	12422	3.03	4.36	7.31	9.02	7.94	8.58
ethernet	19878	0.12	-3.35	-2.58	-2.86	-2.51	17557	-0.36	-1.39	-1.36	-0.76	-1.03	-1.10
vga_lcd	29959	0.49	-1.65	-2.30	-2.72	-2.42	28364	0.14	-0.85	-0.90	-1.12	-1.55	-1.32
des_perf	30491	2.68	5.20	5.77	7.00	7.09	16880	2.33	5.63	5.21	6.49	5.59	5.82
TOTAL	132124	1.59	0.70	0.88	1.34	1.29	105854	1.11	1.55	1.96	2.37	1.92	1.96

Table 5.4: Cut-based restructuring of AND trees, mapped to K-input LUTs

Benchmark	$K = 6$							
	LUTs	% improvement						
		$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$	$k = 7$	$k = 8$
pci_conf_cyc_addr_dec	35	-2.86	-2.86	-2.86	-2.86	-2.86	-2.86	-2.86
steppermotordrive	37	-2.70	0.00	-2.70	-2.70	0.00	0.00	0.00
ss_pcm	102	0.00	-0.98	0.00	0.00	0.00	0.00	0.00
usb_phy	138	2.17	0.72	2.17	2.17	0.72	0.72	0.72
sasc	145	0.00	0.00	0.00	0.00	0.00	0.00	0.00
simple_spi	196	0.51	-0.51	0.51	0.51	0.00	0.00	0.00
pci_spoci_ctrl	216	-3.24	-0.46	2.31	0.93	3.24	-0.46	1.85
i2c	238	-0.84	-1.26	1.26	2.94	0.00	1.68	1.26
systemcdes	494	-0.81	-1.01	1.42	0.81	0.81	-4.86	-6.28
spi	670	8.51	8.06	4.18	4.63	6.42	3.73	8.96
wb_dma	1039	0.29	0.58	0.87	0.96	0.77	0.38	0.10
des_area	941	2.55	-0.32	1.06	-1.49	2.76	1.59	3.40
tv80	1694	-0.18	4.37	5.37	6.38	5.08	3.84	1.77
systemcaes	1821	1.10	-0.22	-1.15	-0.05	0.33	-1.32	-1.48
mem_ctrl	2199	0.77	1.05	2.59	2.91	4.18	3.32	4.46
ac97_ctrl	2870	0.56	0.73	-0.63	-0.66	-0.77	-0.73	-0.38
usb_funct	3103	0.23	1.93	1.19	1.26	1.93	1.58	1.13
pci_bridge32	5082	0.55	3.05	3.76	3.05	3.50	3.94	3.56
aes_core	3128	-6.01	3.71	5.91	3.87	1.44	-8.70	-9.37
wb_conmax	10040	2.53	6.18	10.36	9.72	8.52	8.43	8.87
ethernet	16711	0.80	2.28	5.49	4.70	2.27	2.73	2.69
vga_lcd	26951	0.48	4.98	4.47	4.45	4.09	4.19	4.16
des_perf	12712	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TOTAL	90562	0.54	3.13	4.14	3.83	3.17	2.79	2.81

Table 5.5: Cut-based restructuring of AND trees, mapped to K-input LUTs

Benchmark	Area	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$
pci_conf_cyc_addr_dec	97	0.00	0.00	0.00	0.00	0.00
steppermotordrive	184	17.12	16.85	15.35	15.35	16.98
ss_pcm	408	-15.33	-15.94	-15.45	-15.45	-15.45
usb_phy	454	0.66	0.88	0.72	-0.22	-0.28
sasc	572	-0.13	-1.49	-2.49	-2.49	-2.32
simple_spi	769	1.46	-1.40	-3.35	-3.32	-4.13
pci_spoci_ctrl	976	4.36	3.82	5.25	3.92	4.02
i2c	981	-3.03	-2.29	-3.67	-4.10	-2.73
systemcdes	2724	-0.17	0.20	-0.96	-1.15	-0.61
spi	3304	2.00	1.81	1.21	1.54	1.97
wb_dma	3706	-1.87	-0.49	0.42	-0.44	-0.31
des_area	4437	0.57	-6.45	-4.04	-3.04	-4.85
tv80	7302	2.83	3.79	1.31	1.99	0.93
mem_ctrl	8903	2.38	3.56	7.89	8.83	8.41
systemcaes	9589	-5.78	-6.63	-0.77	-6.42	-6.79
ac97_ctrl	10112	1.94	-3.44	-3.35	-3.53	-3.56
usb_funct	14110	1.86	1.84	1.33	1.51	1.36
pci_bridge32	16030	-0.04	-2.79	-3.90	-3.66	-3.76
aes_core	21662	4.69	4.56	4.07	3.99	4.10
wb_conmax	36430	5.71	5.27	-3.98	-7.12	-5.07
ethernet	52653	0.15	-4.55	-4.76	-4.45	-4.88
vga_lcd	78099	-0.14	-6.96	-6.82	-6.85	-7.03
des_perf	81990	2.07	1.68	1.13	1.04	1.35
TOTAL	355494	1.43	-1.24	-2.17	-2.58	-2.46

Table 5.6: Cut-based restructuring of AND trees, mapped to elements from GSClib

To compare the choices created by our proposed ADD-CHOICES with the existing work, we run the same script three more times, first with `choice` before `map`, then with `transversal` before `map`, and finally with `choicetrans` before `map`. The results are shown in Table 5.7. The column "Base area" shows the same baseline area results as in Table 5.6. The next three pairs of columns show the number of choice nodes created by the corresponding command, and the resulting percent improvement in area after technology mapping. The numbers show that `transversal` generally adds fewer choices than `choice`. Furthermore, the results show that the `map` command in ABC, while being able to use choice nodes, does not typically use them well, since adding flexibility to a problem should never degrade the solution quality.

Since ABC's `map` command does not cope well with choices, we repeat the previous experiment without choice nodes. For any given subject graph with choices, we create 15 subject graphs *without* choices by randomly selecting branches from each of the choice nodes. The results after performing technology mapping on these subject graphs are shown in Table 5.8. For each set of 15 subject graphs, we show the minimum, mean, and maximum standard cell area. The results show that the graphs which follow from the `choice` command generally lead to a smaller standard cell area. In contrast, the graphs following from `transversal` do not. The one exception is vga_lcd: the final results after `transversal` averaged 78098 compared to the baseline of 78099.

Note that the choices generated by `choice` and `transversal` for the benchmark wb_conmax combine well: the results in from applying `choicetrans` are better than from either `choice` or `transversal` alone, as shown in Table 5.7 and Table 5.8. This shows that there is indeed some potential in the use of ADD-CHOICES over the existing `choice` command, albeit little. It is possible that a different technology mapper will be better able to exploit the choices we generate.

5.5 Previous Work with Symmetries

Previous works have used functional symmetries in logic synthesis and/or technology mapping, for example [KD91, KS00]. These were primarily constructive synthesis techniques—they for better or worse completely ignore the structure of the original logical description. Given a function f based on a set of variables, they locate a set of variables X which are symmetric, and re-express f in terms of a $g(X)$, where g has fewer than $|X|$

Benchmarks	Base	choice		transversal		choicetrans	
	area	choices	% imp	choices	% imp	choices	% imp
pci_conf_cyc_addr_dec	97	0	0.00	0	0.00	0	0.00
steppermotordrive	184	16	5.43	0	0.00	16	5.43
ss_pcm	407	27	-0.49	0	0.00	27	-0.49
usb_phy	454	9	0.22	0	0.00	9	0.44
sasc	572	29	-3.15	0	0.00	29	-3.23
simple_spi	768	47	-2.99	0	0.00	47	-2.96
pci_spoci_ctrl	975	71	5.23	0	0.00	71	5.38
i2c	980	76	2.24	0	0.00	76	2.14
systemcdes	2724	222	1.47	2	0.05	224	1.39
spi	3304	293	-0.30	112	-0.17	424	0.28
wb_dma	3706	211	1.16	0	0.13	211	1.20
des_area	4436	356	-7.42	1	0.02	356	-7.27
tv80	7302	500	0.04	48	-0.37	573	0.41
mem_ctrl	8903	640	12.16	60	0.00	695	12.30
systemcaes	9589	222	1.69	79	0.14	1310	0.91
ac97_ctrl	10112	876	-2.61	0	0.00	876	-2.47
usb_funct	14110	1157	0.96	32	0.00	1213	0.95
pci_bridge32	16030	1063	-2.03	116	-1.22	1313	-3.24
aes_core	21662	3387	0.00	173	0.21	3734	-0.08
wb_conmax	36430	3433	-12.72	1261	-7.48	4722	3.81
ethernet	52653	784	-8.39	18	0.00	796	-8.31
vga_lcd	78099	14366	-13.85	4	0.00	14367	-13.84
des_perf	81990	8060	0.63	25	0.00	8073	0.59
TOTAL	355494		-5.28		-0.81		-3.64

Table 5.7: Technology mapping results with choices

Benchmarks	choice			transversal			choicetrans		
	min	mean	max	min	mean	max	min	mean	max
pci_conf_cyc_addr_dec	97	97	97	97	97	97	97	97	97
steppermotordrive	170	174	182	184	184	184	177	179	186
ss_pcm	409	413	415	407	407	407	452	454	456
usb_phy	450	454	456	454	454	454	409	411	414
sasc	570	573	574	572	572	572	571	572	574
simple_spi	759	767	776	768	768	768	760	766	774
pci_spoci_ctrl	940	948	959	975	975	975	938	946	968
i2c	961	967	976	980	980	980	958	968	973
systemcdes	2705	2713	2730	2722	2722	2724	2708	2718	2731
spi	3280	3288	3299	3298	3300	3305	3290	3300	3304
wb_dma	3653	3664	3676	3701	3701	3701	3652	3656	3666
des_area	4437	4442	4453	4435	4435	4436	4424	4444	4455
tv80	7105	7126	7151	7304	7307	7311	7132	7141	7153
mem_ctrl	7815	7882	7926	8898	8902	8908	7825	7875	7902
systemcaes	9546	9565	9604	9569	9583	9587	9986	9995	10000
ac97_ctrl	9989	10002	10006	10112	10112	10112	9559	9573	9596
usb_funct	13813	13850	13877	14103	14107	14109	13830	13863	13876
pci_bridge32	15941	15971	15981	16034	16038	16039	15991	16058	16097
aes_core	21641	21667	21711	21643	21664	21672	21648	21690	21736
wb_conmax	35114	35162	35309	36432	36438	36442	35062	35187	35290
ethernet	52437	52453	52484	52646	52648	52651	52437	52453	52478
vga_lcd	78575	78608	78694	78097	78098	78100	81763	81878	81928
des_perf	81738	81806	81827	81981	81987	81990	78558	78586	78728
TOTAL	352153	352599	353170	355421	355489	355531	352227	352809	353383

Table 5.8: Technology mapping results after removing choices

outputs. The technique does not work as well when there is less symmetry, and [WD98] addresses some of the problems that occur when the function f is not completely symmetric.

5.6 Conclusion

In this chapter we explored the usage of symmetries in the context of technology mapping. Previously, we identified two classes of symmetries, namely those of associative functions and those of prime functions. For each class, we provided an algorithm to permute nodes within a subject graph:

- For associative functions such as ANDs, we calculate the dependencies of each input to the function, and rearrange the inputs so that inputs with common dependencies are adjacent. This reduced the final implementation area for both FPGA and standard cell technologies.
- For a prime function, we calculate the transversal of its symmetry group and add a choice node for the elements in the transversal in order to increase the number of possible matches during mapping to standard cells. Unfortunately, the results are not as encouraging in this case. One reason is that there are not many choices introduced by our algorithm. Another reason is that we depend on nodes with multiple fanouts to create prime functions, but at the same time, a node with multiple fanouts is often a good candidate to form the output of a standard cell, implying that the choices we add may not result in more useful matches to the cell library. Future work should take this into account, for example by performing a preliminary mapping to identify areas where choices can help.

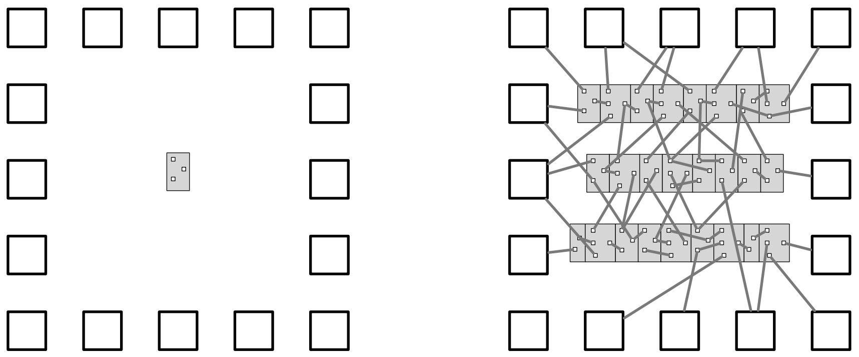
The next chapter applies symmetry to the placement problem in order to reduce wirelength.

6. Placement

After the technology mapping phase in Figure 1.1, comes the placement phase. Most current approaches assume a static circuit netlist, i.e., the connectivity between the gates is assumed to be immutable during placement. This restriction makes the placement problem more tractable but may cause a placer to miss important opportunities for further improvement.

In this chapter, we apply circuit symmetries to improve the results of placement algorithms. Instead of considering all possible symmetries, we restrict the search to two specific forms of symmetry: (1) fully symmetric inputs of tree-based subcircuits, and (2) alternative tree compositions of such subcircuits. We present a polynomial time algorithm that avoids enumerating symmetries and results in an approach that is fast enough to be applied between iterations of a placement algorithm.

This chapter is organized as follows. First, we review the placement problem and basic approaches to solve it. Then, we survey previous works which apply symmetries to optimize placement. Afterwards, we introduce the model and underlying algorithm used in our approach, and show how they are applied in placement. Finally, we present experimental results on a variety of benchmarks and conclude this chapter.


(a) Standard cell in core area, surrounded by terminals

(b) Placed design

Figure 6.1: Standard cell placement arranges cells into rows

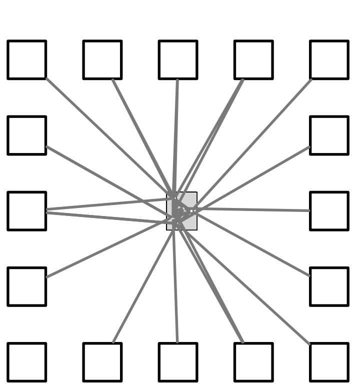
6.1 Introduction

During the placement phase, we are given a **core area** surrounded by a set of fixed terminals, and the netlist from the technology mapping phase. We are tasked to assign cells[1] to non-overlapping locations within the core area, typically in rows. Rather than selecting any arbitrarily legal assignment of locations, we usually aim to optimize some objective, e.g. the total estimated wirelength between cells. Figure 6.1a shows an example of a core area surrounded by terminals; Figure 6.1b shows the same set of terminals interfacing with a placed design. Note that the wires (gray lines) among the cells and terminals denote *estimated* routes for interconnections: actual routes are not determined until the next step of the CAD flow.

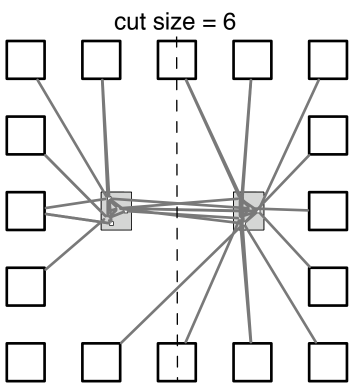
The placement problem itself is typically divided into **global placement**, **detailed placement**, and **legalization**. As the name implies, a global placer produces a rough placement which may not be legal—the other two steps are applied afterwards to obtain a legal placement. Techniques to solve the global placement problem are classified into three groups:

- simulated annealing based on stochastic optimization, e.g. TimberWolf [Sec88].
- analytic placement based on numerical optimization, e.g. [CK84, KSJA91].

[1]We focus on standard cell placement in this chapter because of our choice of experimentation platform. Many of the ideas may be carried over to FPGA placement.



(a) Cells start at center of core area



(b) After bipartitioning

Figure 6.2: One step in an iterative placement algorithm

- partitioning-based placement [Bre77, CKM00] based on recursive partitioning [KL70].

We will use Capo [CKM00], which is an iterative partitioning-based placer, for our experiments, because it is competitive with other placers [Nam06] and its source code is freely available.[2] Therefore, we elaborate further on partitioning-based placement. Placers based on simulated annealing and/or numerical optimization may also be amenable to our approach.

A partitioning-based (global) placer works similarly to the algorithm described in subsection 5.3.1. First, all moveable standard cells are assumed to lie in the center of the core area, as shown in Figure 6.2a. Then, we divide the core area in half, and distribute the cells among the two regions so that the number of connections that cross from one region to the other is minimized. We continue splitting each region this way until the number of cells assigned to a region is small. At any step during this procedure, we may estimate the total wirelength by assuming that each cell is located in the center of its assigned region.

6.2 Previous Work

Previous researchers have used symmetries to optimize a circuit for timing or wirelength, but the algorithms they propose are slow and/or suboptimal.

[2] http://www.openedatools.org/projects/umpack/

The authors of [CHH$^+$04] use symmetries to rewire a circuit for various metrics. Their analysis, based on ATPG techniques, uses logic propagation of noncontrolling values to find wires that may be swapped. They swap one pair of wires at a time using an extension of the algorithm proposed in [Cou97] to optimize the circuit.

The authors of [CMB05a] perform rewiring based on the classical definition of symmetries as permutations of circuit inputs. To perform rewiring among a set of wires, they define a subcircuit using these wires as inputs, and perform an exhaustive search for legal permutations of inputs and outputs at the boundary. Then they enumerate 1000 permutations and select the best rewiring out of these 1000. In order to modify wires *inside* the subcircuit, a new subcircuit must be defined using this new set of wires as inputs. This method suffers from the large number of potential subcircuits, as well as long runtimes to optimize each subcircuit.

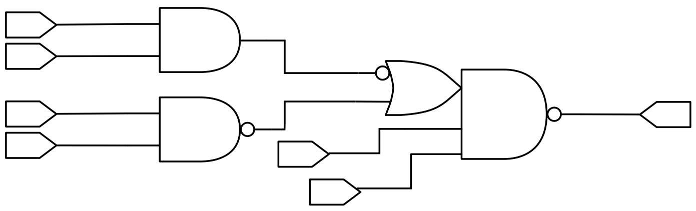
Our proposed method improves on previous work by reducing runtime while preserving quality of results. The key is to efficiently evaluate a set of wire permutations without full enumeration. Given a set of n symmetric wires (for example, the inputs of a parity tree), the previous approaches would potentially have to evaluate $n!$ permutations. In contrast, our approach only requires $O(n^3)$ in runtime.

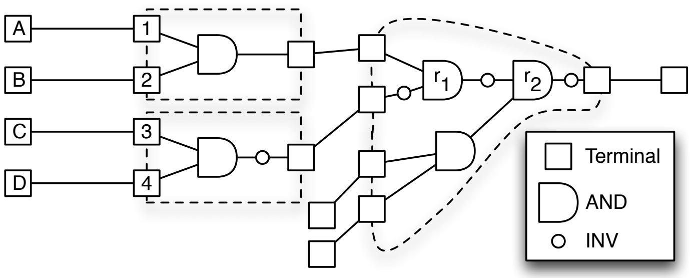
6.3 Preliminaries

6.3.1 Circuit Model

In order to concurrently represent the functionality of a circuit as well as its physical information, we model a circuit by a graph of nodes representing terminals and logic primitives. Each n-input m-output gate is modeled by $n + m$ terminal nodes surrounding logic primitives representing the function of the gate. Terminal nodes may be classified as "source"-type, which are gate outputs and primary inputs, or "sink"-type, which are gate inputs and primary outputs. Throughout this chapter, it is assumed that gates are decomposed into AND and INV primitives; the algorithms can be extended to deal with XOR primitives as well.

The benefit of this model is that we may uniformly reason about both simple gates and complex gates. Figure 6.3 shows a circuit and the corresponding graph which models its functionality. The node labels are used in a later section.


(a) Original circuit



(b) Resulting graph

Figure 6.3: Conversion from circuit to graph

6.3.2 Assignment Problem

The *assignment problem* is an optimization problem with the goal to find a minimum cost matching in a bipartite graph. The classical problem statement is as follows:

> We are given n workers and n jobs, where a cost c_{ij} is incurred if worker i performs job j. Each worker can only perform one job. Find an assignment of workers to jobs such that all jobs are performed and the sum of incurred costs is minimized.

Figure 6.4 shows the cost matrix for a problem with workers {A,B,C,D} and jobs {1,2,3,4}. The optimal solution is to have worker A perform job 4, B do 2, C do 1, and D do 3 for a total cost of 14.

The assignment problem can be solved in $O(n^3)$ time using the Kuhn-Munkres algorithm [Mun57] or one of its variants. The algorithm uses the fact that adding a constant to every entry in a row or every entry in a column does not change the optimal solution, only its cost. First the algorithm tries to employ as many workers as possible for zero cost, i.e. find a maximal zero-cost matching. If the matching is complete, the algorithm terminates. Otherwise, it calculates the minimum additional cost required for a larger matching, adds or subtracts this constant to create more zeros, and restarts (at most n

	1	2	3	4
A	6	4	8	**5**
B	5	**1**	10	4
C	**3**	8	9	6
D	10	2	**5**	6

Figure 6.4: Assignment problem (solution in **bold**)

times). Each iteration of the loop requires time $O(n^2)$. Since the matrix is modified in the algorithm, the actual cost of the optimal assignment is calculated using the original matrix.

6.4 Rewiring Algorithms

In this work, symmetries are identified in a similar manner to that in [CHH⁺04]. Given a circuit graph, we create partitions representing maximal AND-trees as follows:

1. Split the graph at multiple-fanout points, resulting in a set of trees.
2. Split each tree wherever two AND primitives are connected through an odd number of inverters.

This results in trees representing ANDs of maximal size, where some of the inputs may appear inverted. For convenience, we assume that a tree is rooted at an AND node. For example, the graph of Figure 6.3b decomposes into two trees, one rooted at r_1 and one rooted at r_2. Finally, since rewiring can only be performed at terminals, we prune the leaves of each tree to the first sink-type terminal.

From the above decomposition step, we have maximal AND-trees, where each tree is fed by a set of terminals. The terminals of a given tree may be partitioned into two sets according to the number of inverters on the path to the root: a terminal is positive if the number is even, otherwise the terminal is negative. We call this attribute the terminal's *phase*. Now we discuss how to optimize the wiring of each tree.

6.4.1 Leaf Rewiring

Given a tree implementing an AND function, inputs of the same phase may be arbitrarily permuted without changing the function. Thus a minimum cost permutation can be calculated by solving two assignment problems, one for the positive terminals and one for the negative terminals.

The cost is calculated as follows. First, all the sinks are assumed to be disconnected. Then, for source i and sink j, the cost c_{ij} is the increase in size of the bounding box of i if j were added to it.

For example, the tree in Figure 6.3b rooted at r_1 is to be optimized. We calculate the cost of each connection between leaves {1,2,3,4} and their sources {A,B,C,D}, shown in Figure 6.4. After solving the assignment problem, we find that the best rewiring is to connect source A to sink 4, source B to sink 2, and so on.

We argue that this wiring assignment is optimal. First, the Kuhn-Munkres algorithm gives an optimal assignment for a given cost matrix. Second, the cost matrix reflects the true cost of wirelength. The source nets are assumed to be distinct; therefore the cost of any connection is independent of all others and reflects the true increase in wirelength. If the source nets are not distinct, then they can be made so by removing a redundancy.

The algorithm can be adapted for wirelength minimization under timing constraints simply by disallowing connections which decrease timing slack, or by weighting the wirelength costs with timing slack.

6.4.2 Tree Restructuring

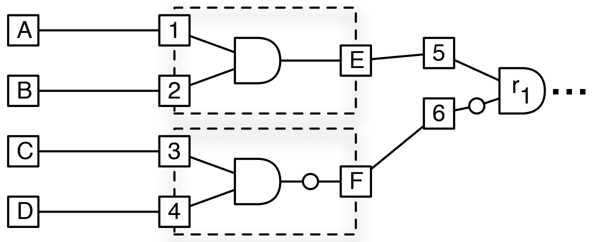
The wirelength of a tree can be further improved by exchanging wires within the tree rather than only the inputs. In this case, we construct an assignment problem encompassing all the terminals, with two constraints:

1. Terminals are matched according to phase.

2. The solution does not contain a cycle.

The first constraint is satisfied in the same way as before, by performing assignment for positive and negative terminals separately. For simplicity, the rewiring problem is presented as a single assignment problem, where invalid assignments have a cost of ∞. However, since the underlying algorithm is in $O(n^3)$, it is more efficient to solve it as two smaller problems.

The cycle-free constraint cannot be easily enforced because the assignment problem assumes that all permutations are feasible, and that costs are independent. Therefore,



(a) Tree to be optimized

	1	2	3	4	5	6
A	6	4	8	5	**3**	∞
B	5	**1**	10	4	10	∞
C	**3**	8	9	6	5	∞
D	10	2	**5**	6	4	∞
E	∞	∞	9	**5**	6	∞
F	∞	∞	∞	∞	∞	**8**

(b) Cost matrix

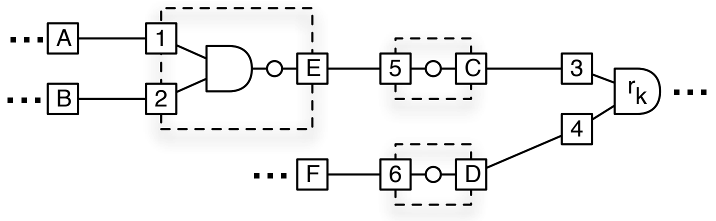
Figure 6.5: Tree restructuring

we have adopted a cycle-avoiding rewiring algorithm that does not necessarily find the best cycle-free permutation, but is guaranteed to be no worse than the previous algorithm.

Given a tree, we start with the original set of gates, and allow any source terminal in the tree to connect to any sink terminal in the tree, with the exception of terminals of different phases, and simple cycles between a gate output and its inputs. Figure 6.5a shows the same tree as before, where the two gates are outlined with dotted boxes. The root r_1 is not considered a gate because its output is not part of the rewiring problem. To avoid a cycle, we forbid a connection between source E and sinks 1 or 2, which is reflected in the cost matrix in Figure 6.5b.

If the solution to the $n \times n$ assignment problem results in a cycle in the graph, we choose an arbitrary source-type terminal node in the cycle and force a connection to its original sink, effectively merging two gates and removing the two terminal nodes. We then solve the new $(n-1) \times (n-1)$ assignment problem. If the solution to the new assignment problem does not lead to a cycle, we have a valid solution. Otherwise, we merge another pair of gates. In the worst case, we revert to the case of leaf rewiring in which we have a single large gate and perform rewiring at the inputs only.

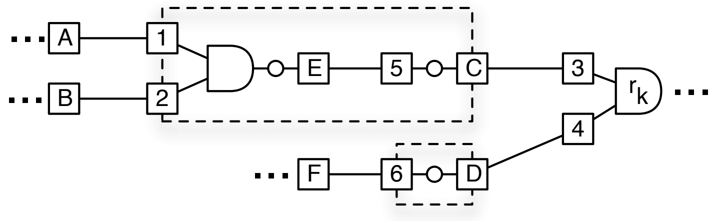
Our experiments confirm that in practical designs, the solution to the assignment problem rarely contains a cycle. Thus, the approach we take is faster than one that would find the best cycle-free solution, with nearly the same results. In the worst case, our tree restructuring degrades to the leaf rewiring problem in the previous section.



(a) Tree to be optimized

	1	2	3	4	5	6
A	5	3	8	**3**	∞	∞
B	4	4	**6**	9	∞	∞
C	3	**5**	10	8	∞	∞
D	**2**	6	8	7	∞	∞
E	∞	∞	∞	∞	4	**3**
F	∞	∞	∞	∞	**4**	4

(b) Cost matrix



(c) Tree after removing cycle

	1	2	3	4	5	6
A	5	3	8	**3**	∞	∞
B	4	**4**	6	9	∞	∞
C	∞	∞	**10**	8	∞	∞
D	**2**	6	8	7	∞	∞
E	∞	∞	∞	∞	**4**	∞
F	∞	∞	∞	∞	∞	**4**

(d) Cost matrix after removing cycle

Figure 6.6: Tree restructuring, avoiding cycles

Consider graph in Figure 6.6a, which shows three gates implementing the function $ab\bar{f}$. The optimum solution to the assignment problem in Figure 6.6b leads to a cycle involving the nodes 1, E, 6, and D. To avoid this cycle, we choose an arbitrary source-type terminal node and force a connection to its original sink. In the example, we force E to connect to 5 by setting $c_{Ej} = \infty$ for all $j \neq 5$ and $c_{i5} = \infty$ for all $i \neq E$. Then we forbid connections between the inputs and output of this new merged gate. The resulting decomposition and cost matrix are shown in Figures 6.6c and 6.6d.

Note that since we consider all feasible restructurings of the tree, we do not need to try different buckets as in [CMB05a, fig. 6]. The algorithm presented in [CMB05a] requires considering multiple overlapping buckets to be able to restructure a tree.

6.5 Iterative Placement with Rewiring

Current global placers assume a fixed netlist structure that must be preserved when placing a design. Instead of keeping a fixed netlist, we propose a placer flow which adaptively restructures the netlist during the placement flow as more information is available. The approach can be used with any placer based on recursive (bi-)partitioning. After each partitioning step, the chip area is sliced into disjoint bins, and the gates within a bin are given temporary locations for the next partitioning step. At this point, we may run one of the proposed rewiring algorithms using the temporary placement.

Unfortunately, current placers are sensitive to small changes in their input. We take the following measures in Capo to avoid disturbing its results:

1. The adjacency list representation of the restructured netlist is kept the same as that of the original netlist wherever possible.

2. Netlist restructuring is done only when the gates are close to their final locations. This is the case in Capo when `CapoPlacer::atBottomLayers()` returns true. Until then, Capo operates in a feedback loop to find a good coarse placement. We found that changing the netlist structure interferes with the heuristics used in the feedback loop.

3. The difference in wirelength from moving a wire from one gate to another in the same bin is assumed to be zero. This is because we do not yet know the final placement of the gates within the bin.

6.6 Experimental Results

To evaluate the two proposed rewiring algorithms, we used instances from the OpenCores subset of the IWLS benchmarks. We obtained legal placements by randomly assigning pad locations along the boundary and running MetaPl/Capo 10.5[3]. We disabled Y-direction whitespace optimization in MetaPl ("-noYFlow") to improve wirelength. Each experiment is performed with an identical set of 15 random seeds for MetaPl, and the average result reported. All experiments on the OpenCores benchmarks are performed on an Intel Xeon E5345 2.33GHz machine, and all rewirings are verified with an equivalence checker.

6.6.1 Post-placement Optimization

Our work differs from previous work in the types of symmetries considered, and in the optimization method. To evaluate the effects of these independently, our first set of experiments compares four different methods for post-placement rewiring:

1. **PhySyn** from UMpack-070919[4], written by the authors of [CMB05a]. This implementation differs from their paper in that it does not consider all possible symmetries, but only simple symmetries such as those that occur in AND and XOR functions. The removal of these optimization opportunities is reported to decrease the optimization potential by about 20%[5].

2. **PhySyn+**, our own implementation of the method described in [CMB05a]. This uses generalized symmetries and exhaustively searches among *all* valid permutations for the best rewiring.

3. **Leaf** Rewiring (§6.4.1).

4. Tree **Restruct**uring (§6.4.2).

Figure 6.7 outlines the differences between the different postprocessors. There is more potential for optimization if generalized symmetries are used instead of simple AND/XOR symmetries, as well as if multiple overlapping subcircuits are optimized instead of disjoint trees. However, both of these require additional runtime to exploit fully. Our first set of experiments evaluates the tradeoff between runtime and solution quality.

[3]http://www.openedatools.org/projects/umpack/
[4]Ibid.
[5]Kai-Hui Chang, personal communication.

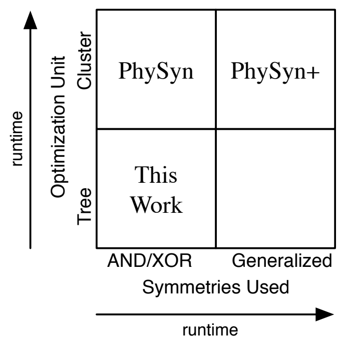


Figure 6.7: Comparison of postprocessors

The results are shown in Table 6.1. Results for postprocessors are shown as percent improvement in wirelength with respect to the final placement. PhySyn is run with two different input size limits for clusters, 8 and 16. PhySyn+ could not complete benchmarks ethernet or vga_lcd within several hours. To conserve space, we report the *fastest* of the three runtimes for the cluster-based algorithms, and the *slowest* of the two runtimes for the algorithms we proposed.

The results for PhySyn and PhySyn+ show that using generalized symmetries will improve wirelength, but not enough to warrant the additional runtime required to find the generalized symmetries. The results for our proposed tree-based rewiring methods show that by more efficiently searching a smaller search space, we can achieve superior results compared to previous work with a >20x improvement in runtime. The improvement in runtime makes it feasible to run our algorithm several times in an iterative placer and thus unlocks more optimization potential.

6.6.2 Integration into Iterative Placer

The next set of experiments examines the effect of integrating symmetry-based rewiring into an iterative placer versus its use purely as a postprocessor. We integrated tree restructuring into MetaPl/Capo as described in §6.5. The results are shown in Table 6.2. The column group "MetaPl/Capo" shows the results for placement followed by symmetry-based rewiring after legalization. The column group "MetaPl/Capo/Rewire" shows the results for placement with integrated rewiring followed by a final pass of rewiring after legalization. The columns marked "%" show the percent improvement in wirelength with respect to MetaPl/Capo.

Benchmark	Placement		Cluster-based Rewiring				Tree-based Rewiring		
	Time (s)	Wirelength	Time (s) (PhySyn-8)	PhySyn-8 %	PhySyn-16 %	PhySyn+ %	Time (s) (Restruct)	Leaf %	Restruct %
steppermotordrive	0.85	1.085e07	0.26	1.367	1.367	1.142	0.04	0.642	0.813
sasc	2.52	3.654e07	0.18	0.479	0.479	0.454	0.05	0.336	0.364
ac97_ctrl	89.73	1.096e09	4.24	0.318	0.318	0.336	0.54	0.262	0.277
usb_funct	100.38	1.501e09	8.20	0.498	0.502	0.527	0.61	0.384	0.472
pci_bridge32	139.69	2.064e09	16.50	0.334	0.338	0.399	0.75	0.203	0.306
aes_core	170.39	2.440e09	21.57	0.498	0.498	0.535	0.90	0.366	0.483
wb_conmax	304.19	6.547e09	50.23	0.271	0.271	0.360	1.68	0.216	0.308
ethernet	718.77	8.223e09	130.36	0.380	0.389	—	3.32	0.306	0.439
des_perf	1157.36	1.196e10	59.81	0.363	0.363	0.372	4.01	0.286	0.348
vga_lcd	2006.90	2.399e10	252.80	0.333	0.340	—	13.52	0.184	0.349
Overall	4690.80	5.786e10	544.15	0.350	0.355	—	25.43	0.241	0.363

Table 6.1: Preprocessing results for OpenCores benchmarks

The results show that rewiring can be integrated into a placer with less than 10% increase in runtime. In all but one benchmark, the integration of rewiring into the placer results in better wirelength. Overall, the benefits of rewiring are additive: MetaPl/Capo/Rewire+postproc gives better results than MetaPl/Capo+postproc. The reason is that integration into a placer allows for the optimal placement of a fluid netlist structure, rather than the placement of a *fixed* netlist structure.

6.6.3 Timing-driven Rewiring

The next set of experiments explores the efficacy of rewiring as a postplacement speed-up technique where certain wires are deemed "critical" and others are not. We ran a static timing analysis tool (OAGear Timer[6]) and marked the top 5% of wires with most negative timing slack as critical. Rewiring is performed on the same placements as before, this time to minimize the *weighted* wirelength, where critical wires have a weight of 20 and all others have a weight of 1. The results are shown in Table 6.3. Wirelengths for noncritical and critical wires are shown separately. In most cases, rewiring is able to focus more effort on the critical wires, and is thus useful as a speed-up technique.

The final set of experiments assumes that wire criticality is known before placement, and that therefore the placer can be given weights for all the wires. Placement and rewiring are performed to minimize the weighted wirelength, using the same set of weights as in the previous timing-driven experiment. The results are shown in Table 6.4 with the same placer set-ups as before. Again, percent improvements are shown with respect to MetaPl/Capo. The results indicate that rewiring is able to achieve large improvements in wirelength for critical wires with no degradation of noncritical wires. As before, the integration of rewiring into the placer causes problems for the smallest benchmark. Overall, however, the results again indicate that rewiring achieves larger improvements when integrated into the placer than when simply run as a postprocessor. Compared to Table 6.3, MetaPl/Capo will shorten critical wires and lengthen noncritical wires as expected when given wire weights. Furthermore, the reduction in HPWL for critical wires is obviously greater than what rewiring alone can achieve, since we alternately modify placement and netlist structure. However, this actually increases the optimization potential of rewiring for both noncritical *and* critical wires. Thus, symmetry-based rewiring is

[6]http://www.openedatools.org/projects/oagear/

	MetaPl/Capo			MetaPl/Capo/Rewire			
	Placement		+ postproc	Placement			+ postproc
Benchmark	Time (s)	HPWL	%	Time(s)	HPWL	%	%
steppermotordrive	0.85	1.085e07	0.814	0.91	1.085e07	-0.038	0.493
sasc	2.52	3.654e07	0.364	2.63	3.643e07	0.295	0.541
ac97_ctrl	89.73	1.096e09	0.277	95.16	1.095e09	0.119	0.315
usb_funct	100.38	1.501e09	0.472	104.99	1.497e09	0.286	0.550
pci_bridge32	139.69	2.064e09	0.306	147.85	2.057e09	0.318	0.474
aes_core	170.39	2.440e09	0.483	182.65	2.428e09	0.477	0.730
wb_conmax	304.19	6.547e09	0.308	314.21	6.532e09	0.236	0.426
ethernet	718.77	8.223e09	0.514	757.78	8.171e09	0.631	0.807
des_perf	1157.36	1.196e10	0.348	1242.36	1.194e10	0.167	0.396
vga_lcd	2006.90	2.399e10	0.350	2163.93	2.378e10	0.892	1.002
Overall	4690.80	5.786e10	0.374	5012.46	5.754e10	0.562	0.728

Table 6.2: Integration of rewiring into MetaPl/Capo

	Placement		+ postproc	
Benchmark	Noncrit HPWL	Crit HPWL	Noncrit %	Crit %
steppermotordrive	1.039e07	4.526e05	0.786	1.157
sasc	3.506e07	1.479e06	0.367	0.216
ac97_ctrl	1.035e09	6.043e07	0.260	0.451
usb_funct	1.391e09	1.103e08	0.468	0.391
pci_bridge32	1.916e09	1.475e08	0.257	0.671
aes_core	2.302e09	1.378e08	0.436	0.917
wb_conmax	5.799e09	7.481e08	0.201	0.748
ethernet	7.750e09	4.733e08	0.331	1.044
des_perf	1.072e10	1.239e09	0.330	0.377
vga_lcd	2.216e10	1.826e09	0.151	1.909
Overall	5.312e10	4.744e09	0.246	1.118

Table 6.3: Timing-driven postplacement rewiring using weighted wires

equally applicable whether wire weights are given to the placer or not.

6.6.4 Industrial Designs

To evaluate our tree restructuring algorithm, we also performed postplacement rewiring on a variety of industrial designs. We were unable to run MetaPl/Capo on these designs, and therefore only present postplacement results. The designs were synthesized and placed with a commercial tool; we ran our algorithm afterwards on the placed designs on an 3.2GHz Intel Xeon processor. Column 5 in Table 6.5 shows the improvement in wirelength assuming all wires have unit weight. We also ran our algorithm giving higher weight (20) to the 5% most critical wires. Column 10 shows the improvement in wirelength for critical wires. As the results show, our algorithm is scalable to very large designs; we expect that integration into a placer will result in even better wirelength improvement.

6.7 Conclusion

In this chapter we presented an approach to wirelength optimization based on exploiting functional symmetries, in which a set of wires may be permuted without changing func-

	MetaPl/Capo				MetaPl/Capo/Rewire			
	Placement		+ postproc		Placement		+ postproc	
Benchmark	Noncrit HPWL	Crit HPWL	Noncrit %	Crit %	Noncrit %	Crit %	Noncrit %	Crit %
steppermotordrive	1.086e07	2.789e05	0.837	2.314	0.379	-6.087	0.763	-3.843
sasc	3.828e07	1.025e06	0.305	0.341	0.056	1.566	0.263	1.840
ac97_ctrl	1.147e09	2.483e07	0.248	1.067	0.136	0.348	0.310	1.084
usb_funct	1.560e09	5.734e07	0.396	0.686	0.233	0.173	0.479	0.731
pci_bridge32	2.269e09	4.627e07	0.317	1.813	0.553	0.821	0.678	2.129
aes_core	2.548e09	4.806e07	0.435	1.609	0.440	0.932	0.666	2.053
wb_conmax	7.147e09	3.656e08	0.260	0.371	0.157	0.102	0.320	0.396
ethernet	9.255e09	2.779e08	1.039	1.401	1.550	1.292	1.736	1.833
des_perf	1.543e10	3.807e08	0.243	1.116	0.091	0.667	0.255	1.556
vga_lcd	2.966e10	7.001e08	4.067	3.119	5.593	4.746	5.909	5.937
Overall	6.907e10	1.902e09	2.007	1.767	2.688	2.142	2.924	2.981

Table 6.4: Integration of rewiring into MetaPl/Capo with weighted wires

Benchmark		Rewiring			Rewiring with weighted wires				
Name	Cells	HPWL	Time (s)	HPWL (%)	Noncrit HPWL	Crit HPWL	Time (s)	Noncrit %	Crit %
8625	6046	2.381e08	0.98	0.98	2.310e08	7.119e06	1.09	0.91	2.21
f49d	8545	3.800e08	1.12	1.12	3.568e08	2.318e07	1.19	1.00	2.49
d556	32677	5.181e09	4.66	0.26	5.036e09	1.454e08	5.57	0.26	0.17
31f9	34797	4.368e09	5.78	0.43	4.055e09	3.129e08	7.14	0.41	0.44
6107	46552	6.362e09	5.35	0.85	6.014e09	3.483e08	6.87	0.78	1.54
8743	62663	2.850e09	6.51	0.36	2.772e09	7.768e07	7.87	0.33	1.43
2363	63757	1.302e10	15.20	0.16	1.258e10	4.421e08	16.62	0.16	0.04
0abf	72053	2.819e09	9.99	0.86	2.585e09	2.344e08	11.89	0.79	1.29
c597	100733	1.049e10	16.23	0.70	1.019e10	3.077e08	18.95	0.61	2.89
7ea7	107871	1.660e10	11.14	0.27	1.583e10	7.710e08	13.45	0.24	0.80
7744	138417	1.029e10	19.38	0.44	9.901e09	3.878e08	22.45	0.40	1.14
94d5	153100	3.286e10	23.33	0.27	3.166e10	1.202e09	28.50	0.12	2.36
ba19	206898	3.213e10	24.48	0.27	3.061e10	1.520e09	30.16	0.25	0.59
b385	222167	8.550e09	17.07	0.64	7.677e09	8.728e08	20.50	0.46	1.89
abb0	363827	1.858e10	35.90	0.40	1.710e10	1.484e09	42.04	0.39	0.53
c67e	367236	6.634e10	54.40	0.47	6.078e10	5.558e09	64.91	0.47	0.37
bc04	442510	3.420e10	27.32	0.09	3.058e10	3.625e09	35.37	0.09	0.02
Overall		2.653e11	278.83	0.36	2.479e11	1.732e10	334.55	0.32	0.66

Table 6.5: Rewiring results for industrial designs

tionality. The symmetries we used are based on properties of AND trees, namely that any acyclic restructuring of a tree implements the same function. These represent a large subset of the total set of symmetries, and permit solution with polynomial time algorithms. The efficiency of our algorithms permits us to integrate tree restructuring into an iterative placer. The results demonstrate that the combination achieves better wirelength than the two separately, with a small runtime penalty.

As a post-processing step, our proposed algorithms offer a fast and consistent improvement in results. As an optimization step between placement iterations, our algorithms allow the placer to implicitly explore a larger solution space, and with the exception of the smallest benchmark, obtain better results.

In the next chapter, we address the use of symmetries in Boolean satisfiability.

7. Boolean Satisfiability

The SAT problem is an important $\mathcal{NP}$-complete problem, with applications in optimization and verification. Certain examples are known to require exponential time to solve using the resolution principle [Urq87], but only polynomial time if we use the principle of symmetry [Kri85]. Previous work introduces symmetry breaking predicates which allows us to use the symmetry principle with an off-the-shelf SAT solver. We improve on previous work by identifying key properties of symmetry breaking predicates and provide an algorithm to obtain strong predicates. Experimental results show that our algorithm is efficient and leads to faster SAT solution times.

This chapter is organized as follows. First, we illustrate the basic idea behind symmetry-breaking in SAT, and show where efficiency improvements are to be expected. Then, we summarize the approach taken in previous work to break symmetries, and describe our improvements to enhance the effect of symmetry-breaking. Finally, we compare the solution times for various SAT instances using different techniques for symmetry.

7.1 Introduction

Boolean satisfiability (SAT) is a well-studied $\mathcal{NP}$-complete problem, permitting efficient reductions from problems in optimization and verification [BCCZ99]. Modern SAT solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) procedure [DP60, DLL62], which uses the resolution rule to find a solution to a CNF formula or prove unsatisfiability.

By using a conflict-based mechanism to choose which resolution steps to apply, modern SAT solvers are able to find solutions quickly to wide variety of problems [ZMMM01].

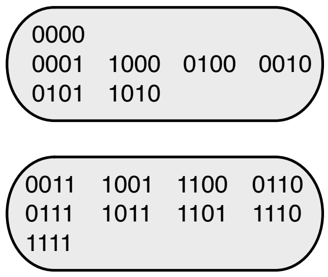
However, certain problems cannot be solved using a polynomial number of resolution steps, for example instances of the pigeonhole problem [Hak85] for any fixed value of n. The pigeonhole principle states that $n+1$ pigeons cannot be placed into n holes without two pigeons sharing a hole. The author of [Urq87] provides other examples of problems which require exponential-size proofs.

Using the principle of symmetry, many formulas can be proven with exponentially shorter proofs [Kri85]. For example, to prove the pigeonhole principle for some value of n, we assume that pigeon 1 is placed into hole 1, as well as an inductive hypothesis that n pigeons cannot be placed into $n-1$ holes. To complete the proof, we can argue that the result would be the same no matter which hole pigeon 1 is placed in—the n cases are symmetric. The following example shows how symmetries can be applied for another SAT formula.

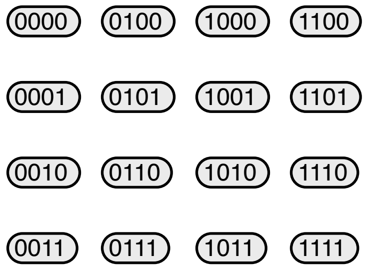
Example 7.1. Let h be the function $x_1x_2 + x_2x_3 + x_3x_4 + x_4x_1$. Assuming that we are interested in solving the satisfiability problem, we can divide $\mathbb{B}^4$ into two partitions: those points in $\mathbb{B}^4$ for which h evaluates to false, and those for which h evaluates to true. Given an oracle which provides points from any specified partition (but doesn't know which is which), we can simply ask the oracle for a point from each (Figure 7.1a) to solve the SAT problem for h. Without such an oracle, we assume that each minterm lies in its own class (Figure 7.1b), and use the DPLL procedure. (Note that in this chapter, x_1 is shown as the leftmost bit in any minterm.)

Symmetry allows us to establish a different partition. Let H denote h's symmetries $Sym(h) = \langle\{(1,2,3,4)\}\rangle = \{(), (1,2,3,4), (1,3)(2,4), (1,4,3,2)\}$. Recall that by definition, $h(x) = h(x^\pi)$ for any $\pi \in H$. Then given $x = 0001$ and $\pi = (1,2,3,4)$, x and $x^\pi = 1000$ may be placed into one equivalence class, since $h(x) = h(x^\pi)$. By exhausting all permutations from H, we define the partitions shown in Figure 7.1c. Thus, to check satisfiability for h, it suffices to check $h(x)$ for $x \in \{0000, 0100, 1001, 1010, 0111, 1111\}$ or for any other six representatives from the equivalence classes.

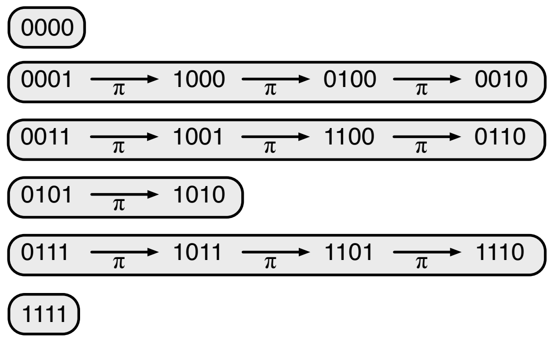
Given a SAT formula and its symmetries, the problem we are looking to solve is the enumeration of minterms from each of the equivalence classes. One way to do this is to extend a constraint solver to handle a "symmetry rule" in addition to a resolution rule



(a) Coarsest equivalence classes (h and $\bar{h}$)



(b) Finest equivalence classes (minterms)


0000
0001 π 1000 π 0100 π 0010
0011 π 1001 π 1100 π 0110
0101 π 1010
0111 π 1011 π 1101 π 1110
1111

(c) Classes formed from symmetry $\pi = (1, 2, 3, 4)$

Figure 7.1: Equivalence classes for the satisfiability problem of $h = x_1x_2 + x_2x_3 + x_3x_4 + x_4x_1$

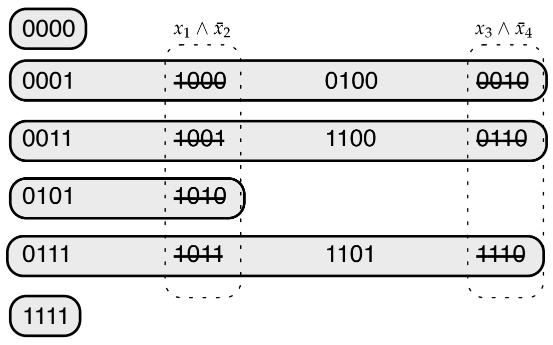


Figure 7.2: SBP $(\bar{x}_1 \vee x_2) \wedge (\bar{x}_3 \vee x_4)$ removes $\frac{7}{16}$ of the search space

and any other inference rules [BFP89a]. This approach is direct and effective, but such an approach may not be able to exploit ongoing advances in SAT solvers.

An alternative approach is to encode symmetries as ordinary constraints to be given to an off-the-shelf SAT solver. These so-called "symmetry breaking predicates" (SBPs) prevent the solver from exploring the entirety of the search space by evaluating to false for some (but not all) elements from each equivalence class. The runtime savings comes from the assumption that the original formula evaluates to false for some of these same minterms as well, though the solver takes longer to show this for the original SAT formula than it does to evaluate the SBPs—the SBPs provide a shortcut to bypass portions of the search space.

Example 7.2. Given a complex SAT formula ϕ with the same symmetries as before ($Sym(\phi) = \langle \{(1,2,3,4)\} \rangle$), we can instead provide the SAT solver with a new formula $\phi' = \phi \wedge \psi$ where $\psi = (\bar{x}_1 \vee x_2) \wedge (\bar{x}_3 \vee x_4)$. The SBP ψ is valid because it permits at least one element from each equivalence class (Figure 7.2). If we assume a simplistic model of the SAT solver whereby solution time is linear with respect to the size of the search space, we can conclude that our SBP reduces solution time by $\frac{7}{16}$.

The penalty of the SBP approach is that many predicates may be required to break *all* symmetries of a formula, which defeats the purpose of using SBPs—the basic premise is that it is more efficient for a SAT solver to process the SBPs than it is to process the original formula. Thus, there is a trade-off between emitting too many SBPs (causing a corresponding slowdown in the SAT solver), or emitting too few and allowing the SAT solver to perform excess search.

The SBP approach consists of three steps: 1) finding the set of symmetries, 2) identifying good symmetries to break, and 3) encoding the symmetries as CNF constraints. With recent advances in solving graph automorphism [DLSM04, DSM08], the first step is effectively solved. The last step is also a solved problem [ASM06]. Our work focuses on the second step, for which no clear strategy has been proposed. We develop a procedure to find a small set of symmetries, which when broken, remove a comparatively large portion of the search space.

7.2 Preliminaries and Previous Work

The basic framework for constructing symmetry breaking predicates was first described in [CGLR96]. The task is divided into three subproblems:

1. find automorphisms/symmetries G of a CNF formula, using for example Nauty [McK81] or Saucy [DLSM04]
2. select a set of permutations in G
3. emit predicates for those permutations

The authors propose three methods to select symmetries: 1) choose generators from Nauty, 2) choose nodes from a "symmetry tree", and 3) arbitrarily. Given a permutation π and an ordering of the variables $V = x_1, x_2, \ldots$ the following set of constraints breaks symmetries between variable assignments by imposing a lexicographical ordering among points in $\mathbb{B}^n$:

$$\begin{array}{rcl} & & v_1 \leq v_1^\pi \\ v_1 = v_1^\pi & \Longrightarrow & v_2 \leq v_2^\pi \\ (v_1 = v_1^\pi) \wedge (v_2 = v_2^\pi) & \Longrightarrow & v_3 \leq v_3^\pi \\ & \vdots & \end{array}$$

where $x \leq y$ is shorthand for $(\bar{x} \vee y)$. Using the same ordering of the variables, we emit a set of constraints for each permutation as desired. The constraints for any given permutation are kept linear in length by introducing auxiliary variables e_i:

$$e_1 \wedge \bigwedge_i \left((e_i \Longrightarrow v_i \leq v_i^\pi) \wedge (e_i \wedge (v_i = v_i^\pi) \Longrightarrow e_{i+1})\right)$$

We can simplify further using the fact that $(x \leq y)$ implies that $(x = y) \equiv (x = 1 \vee y = 0)$.

Afterwards, [ARSM03] extends the approach to support phase shift symmetries, i.e. symmetries that involve the swapping of literals with different polarities. Thus the symmetry $(x_0, \bar{x}_0)(x_1, x_2)$ of a multiplexer $x_0 x_1 + \bar{x}_0 x_2$ is supported. The authors of [ASM06] describe various optimizations which apply if, for example, $l_i^\pi = l_i$ or $l_i^\pi = \bar{l}_i$. As a consequence, the size of the SBP is linear with respect to the number of moved points of π.

The authors of [LR02] and [Roy07] present theoretical bounds on the number of predicates required to break *all* symmetries, assuming G satisfies certain properties (which may not necessarily be the case). The practical application of that work is limited, because the number of additional predicates may outweigh the benefit of breaking all symmetries, since the runtime of a SAT solver is dependent on the number of predicates.

The authors of [KP08] present a complete characterization of the symmetry-induced equivalence classes for formulas which consist entirely of so-called packing or partitioning constraints. In contrast, we consider SAT formulas which may consist of arbitrary clauses.

7.3 Strong Symmetry Breaking Predicates

For a permutation π and sequence V, let $P(\pi, V)$ denote the symmetry breaking predicate as defined previously, where the variables are assumed to be ordered according to V.

$$P(\pi, V) = \bigwedge_{i=1}^{n} \left(\bigwedge_{j=1}^{i-1} (v_j = v_j^{\pi}) \implies v_i \leq v_i^{\pi} \right)$$

It is assumed that we will use auxiliary variables to achieve a linear-length encoding, but they are not shown to simplify the presentation. As before, we adopt the notational convenience whereby we use the same symbol to denote a function/expression as well as the set of minterms in $\mathbb{B}^n$ for which the expression evaluates to true.

To reiterate the objective of this chapter, our goal is: given a formula f and its symmetries $Sym(f)$, determine a set of permutations $T \subseteq Sym(f)$ and variable ordering V such that $P(T, V)$ consists of a small number of constraints and $|P(T, V)|$ is small. The following conjecture directs the heuristics which we apply toward this problem:

Conjecture. *Given a set of permutations T over the set of points $\{1, 2, \ldots, n\}$, let G denote the group generated by T: $G = \langle T \rangle$. Suppose that there exists a sequence of points β such that β and T form a base and strong generating set (SGS) for G. Then there exists an ordering V such that V and T form a base and SGS for G, and $|P(T, V)|$ is minimal over all $n!$ orderings.*

Thus, one possible (but very inefficient) strategy for solving our problem is to arbitrarily choose some $T \subseteq Sym(f)$ and then find an ordering V which makes T an SGS. The next example illustrates the effect of variable ordering on the efficiency of SBPs—given two different variable orderings V and V', $|P(T, V)| < |P(T, V')|$.

Example 7.3. Let $T = \{(1,3),(1,2)(3,4)\}$, representing the symmetries of function $x_1x_3 + x_2x_4$. It can be shown that the equivalence classes of $\mathbb{B}^4$ under $\langle T \rangle$ are the same as those shown in Figure 7.1c, i.e. there are six classes.

Under the variable ordering of $V = x_2, x_1, x_3, x_4$, T forms an SGS with stabilizer chain $\langle T \rangle \supseteq \langle \{(1,3)\} \rangle \supseteq \{()\}$. $P(T,V)$ consists of the following constraints:

$$(x_1 \leq x_3)(x_2 \leq x_1)(x_2 = x_1 \Longrightarrow x_3 \leq x_4)$$

The only points in $\mathbb{B}^n$ which satisfy $P(T,V)$ are the six minterms 0000, 0001, 0011, 1010, 1011, and 1111—one for each class. In other words, $|P(T,V)|$ is minimal among the 24 orderings of x_i.

Assuming a different ordering $V' = x_1, x_2, x_3, x_4$, we arrive at the following constraints $P(T,V')$:

$$(x_1 \leq x_3)(x_1 \leq x_2)(x_1 = x_2 \Longrightarrow x_3 \leq x_4)$$

It can be verified that $|P(T,V')| = 8$, with $P(T,V')$ consisting of the minterms 0000, 0001, 0011, 0100, 0101, 0110, 0111, 1111.

Since we know of no efficient algorithm to determine a base which makes a generating set a *strong* generating set, our strategy is to proceed in the opposite direction—first determine a variable ordering V, then find an SGS T, and finally refine our choices for T and V.

7.3.1 Motivation for Strengthening

Before proceeding further, it would be prudent to evaluate the output of an automorphism finder (step 1 of the SBP approach) to determine whether it can be used directly to construct effective SBPs.

In [ARSM03], the authors acknowledge that certain generators are better than others, e.g. $\{(1,2), (2,3), (3,4)\}$ are better than $\{(1,2), (1,2,3,4)\}$. They note that the software program they use (Saucy) is “lucky” and produces good generators for T. One can verify that this does not always occur, for example if the order of the variables are changed.

Example 7.4. Consider the following instance of the pigeonhole problem for $n = 3$. Pigeon i is in hole j iff variable $x_{3(j-1)+i}$ is true.

$$\begin{array}{rrrr}
(x_1 \vee x_2 \vee x_3) & (\bar{x}_1 \vee \bar{x}_4) & (\bar{x}_2 \vee \bar{x}_5) & (\bar{x}_3 \vee \bar{x}_6) \\
(x_4 \vee x_5 \vee x_6) & (\bar{x}_1 \vee \bar{x}_7) & (\bar{x}_2 \vee \bar{x}_8) & (\bar{x}_3 \vee \bar{x}_9) \\
(x_7 \vee x_8 \vee x_9) & (\bar{x}_1 \vee \bar{x}_{10}) & (\bar{x}_2 \vee \bar{x}_{11}) & (\bar{x}_3 \vee \bar{x}_{12}) \\
(x_{10} \vee x_{11} \vee x_{12}) & (\bar{x}_4 \vee \bar{x}_7) & (\bar{x}_5 \vee \bar{x}_8) & (\bar{x}_6 \vee \bar{x}_9) \\
 & (\bar{x}_4 \vee \bar{x}_{10}) & (\bar{x}_5 \vee \bar{x}_{11}) & (\bar{x}_6 \vee \bar{x}_{12}) \\
 & (\bar{x}_7 \vee \bar{x}_{10}) & (\bar{x}_8 \vee \bar{x}_{11}) & (\bar{x}_9 \vee \bar{x}_{12})
\end{array}$$

By providing the graphical representation of this CNF to Saucy, we obtain the generators $T = \{(1,4)(2,5)(3,6),\ (1,2)(4,5)(7,8)(10,11),\ (2,3)(5,6)(8,9)(11,12),\ (4,7)(5,8)(6,9),\ (7,10)(8,11)(9,12)\}$. Assuming $V = x_1, x_2, \ldots, x_{12}$, $P(T,V)$ is satisfied by only 87 minterms, which is a large reduction from the 4096 minterms in $\mathbb{B}^{12}$.

If we rewrite the CNF using a different assignment of variables for pigeons, i.e. using an ordering $V' = x_3, x_7, x_{12}, x_{11}, x_1, x_6, x_9, x_2, x_8, x_5, x_4, x_{10}$

$$\begin{array}{rrrr}
(x_3 \vee x_7 \vee x_{12}) & (\bar{x}_3 \vee \bar{x}_{11}) & (\bar{x}_7 \vee \bar{x}_1) & (\bar{x}_{12} \vee \bar{x}_6) \\
(x_{11} \vee x_1 \vee x_6) & (\bar{x}_3 \vee \bar{x}_9) & \ldots & \\
(x_9 \vee x_2 \vee x_8) & (\bar{x}_3 \vee \bar{x}_5) & \ldots & \\
(x_5 \vee x_4 \vee x_{10}) & \ldots & &
\end{array}$$

then we obtain a different set of generators from Saucy: $T' = \{(1,12,11,7,6,3)(2,10,9,4,8,5),\ (1,11)(2,9)(3,7)(4,5),\ (1,6)(2,8)(4,10)(7,12),\ (1,2)(6,8)(9,11),\ (2,4)(5,9)(8,10)\}$. This time, Saucy is not as "lucky" as before. Using either variable ordering V or V' yields inferior SBPs, since $|P(T',V)| = 394$ and $|P(T',V')| = 122$.

This illustrates the need for algorithms to determine good values for T and V. In the next subsection, we explore and model the effects that different permutations and variable ordering have on the efficiency of SBPs.

7.3.2 Desired Traits of SBPs

Besides the property of V and T forming a base and SGS, two traits of $P(T,V)$ contribute to larger reductions in search space. These are: transitivity among the inequalities, and small sets of moved points for each associated permutation.[1]

Transitivity

Transitivity among inequalities, i.e. constraints of the form $(x_1 \leq x_2)(x_2 \leq x_3)$, increases the pruning power over constraints with no transitivity, i.e. $(x_1 \leq x_2)(x_1 \leq x_3)$. The first pair permits 4 minterms from $\mathbb{B}^3$ while the second pair permits 5. The difference grows exponentially as there are more constraints—the number of minterms in $\mathbb{B}^k$ which satisfy $k-1$ transitive constraints is $k+1$

$$\left|\bigwedge_{i=2}^{k}(x_{i-1} \leq x_i)\right| = k+1$$

while the number of minterms which satisfy an equal number of non-transitive constraints is $2^{k-1}+1$

$$\left|\bigwedge_{i=2}^{k}(x_1 \leq x_i)\right| = \left|\bigwedge_{i=2}^{k}(x_i \leq x_1)\right| = 2^{k-1}+1$$

The cost model can be generalized from simple "serial" and "parallel" sets of inequalities to those which form a tree. Given a set of inequalities $x_i \leq x_j$, we construct a graph with a directed edge (i,j) for each inequality $x_i \leq x_j$. Assuming that the graph is a tree, the following rules describe how to compute $|\bigwedge(x_i \leq x_j)|$.

Proposition 1. *Given a tree of n nodes representing a system of inequalities over variables in $\mathbb{B}$, the number of points in $\mathbb{B}^n$ which satisfy the inequalities is $p(r)$, assuming the tree is rooted at r:*

- *For a leaf node i, $p(i) = 2$.*
- *For a non-leaf node i, $p(i) = \Pi_{j \in child(i)} p(j) + 1$*

Proof. Since each node represents a fresh Boolean variable, each leaf node represents the two minterms $\{0,1\}$ in its own independent $\mathbb{B}^1$ space.

For a non-leaf node i, there are two possible cases: $x_i = 0$ and $x_i = 1$. When $x_i = 0$, each of i's children is an independent Boolean space unconstrained by x_i, and

[1]One may argue that these are a byproduct of the property of having a base and SGS.

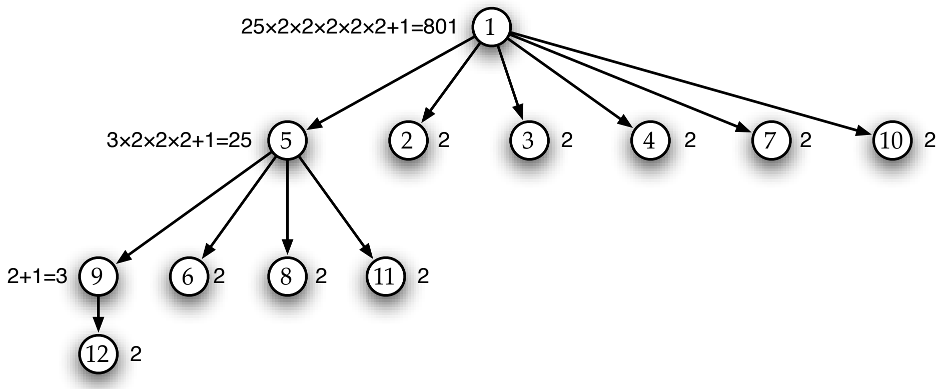


Figure 7.3: A tree of inequalities which permits 801 minterms from $\mathbb{B}^{12}$

the values for each of variables represented by i's descendants may be drawn from its respective space. However, if $x_i = 1$ then the variables for all of i's descendants must be 1 as well. □

Example 7.5. The following set of inequalities

$$P = \left(\bigwedge_{i \in \{5,2,3,4,7,10\}} (x_1 \leq x_i) \right) \wedge \left(\bigwedge_{i \in \{9,6,8,11\}} (x_5 \leq x_i) \right) \wedge (x_9 \leq x_{12})$$

is represented graphically by the tree in Figure 7.3. By applying the above set of rules, we find that $|P| = p(1) = 801$.

Moved Points

The other factor which affects the quality of $P(T, V)$ is the set of moved points for each $\pi \in T$.[2] Assuming the variable ordering $V = x_1, x_2, \ldots, x_n$, the fraction of $\mathbb{B}^n$ pruned by the constraint

$$\bigwedge_{i=1}^{k-1} (x_i = x_i^\pi) \Longrightarrow x_k \leq x_k^\pi$$

within the predicate $P(\pi, V)$ (assuming that $x_k \neq x_k^\pi$) depends on the number of moved points among $\{x_1, x_2, \ldots, x_{k-1}\}$.

[2]Recall that the set of moved points for π is the set $\{i : i^\pi \neq i\}$.

Proposition 2. *Any irredundant set of k equalities of the form $x_i = x_j$ constrains $\mathbb{B}^n$ space by a factor of* 2^{-k}.

Proof. Note that a cycle of equalities $x_1 = x_2, x_2 = x_3, \ldots, x_l = x_1$ contains one redundant equality, and constrains Boolean space by 2^{1-l}. A set of k irredundant equalities either forms a cycle of length $k+1$, or a set of m disjoint cycles whose lengths add to $k+m$. □

Thus, each successive constraint in the series $P(\pi, V)$ prunes an ever decreasing portion of Boolean space, since each constraint is predicated on a larger set of equalities.

Example 7.6. Suppose $\pi = (1,2)(4,3,5)$ and $V = x_1, x_2, x_3, x_4, x_5$. The predicate $P(\pi, V)$ consists of the following constraints which individually prune some fraction of $\mathbb{B}^n$:

$x_1 \le x_2$	$1/4$ of $\mathbb{B}^n$
$(x_1 = x_2) \Longrightarrow x_2 \le x_1$	0 (end of cycle)
$(x_1 = x_2) \Longrightarrow x_3 \le x_5$	$1/8$
$(x_1 = x_2)(x_3 = x_5) \Longrightarrow x_4 \le x_3$	$1/16$
$(x_1 = x_2)(x_3 = x_5)(x_4 = x_3) \Longrightarrow x_5 \le x_4$	0

Altogether, $P(\pi, V)$ prunes $7/16$ of Boolean space.

Proposition 3. *For any permutation π and variable ordering V, the predicate $P(\pi, V)$ prunes Boolean space by at most* $1/2$.

Proof. Suppose π does not map any variable x_i to its negation $\bar{x}_i$. Since each constraint within $P(\pi, V)$ is orthogonal, and each successive constraint has half the pruning power of the previous one, we have the infinite series $\sum_{i=2}^{\infty} 2^{-i} = 1/2$. In particular, if π moves m points and consists of c cycles, then $|P(\pi, V)| = (1/2 + 1/2^{1+m-c})2^n$.

If π maps a variable to its negation, then $P(\pi, V)$ contains a constraint

$$\bigwedge_{i=1}^{k-1} (x_i = x_i^{\pi}) \Longrightarrow x_k \le \bar{x}_k$$

and $|P(\pi, V)|$ is precisely $(1/2)2^n$. □

These lead us to conclude on two "rules of thumb" regarding T and V. First, it is beneficial to reduce the number of moved points for each permutation π, because each additional constraint in $P(\pi, V)$ leads to diminishing returns. Second, for any variable x_j, there should exist some π such that $x_j \neq x_j^\pi$ and that there are few moved points that precede x_j in the ordering V, i.e. $\{x_i : (x_i \neq x_i^\pi) \wedge (x_i \prec_V x_j)\}$ is small. This produces the constraint

$$\bigwedge(\text{small number of equalities}) \implies x_j \leq x_j^\pi$$

for each x_j, and more effectively distributes the pruning effect for the permutations in T.

7.4 Algorithms

In this section describe four steps which may be applied in order to derive a T and V for which $P(T, V)$ consists of relatively few constraints, for for which $|P(T, V)|$ is small.

1. construct Jerrum branching
2. order base variables to maximize chaining
3. simplify permutations
4. establish ordering among non-base variables

7.4.1 Jerrum Branching

The first step to finding good values for $T \subseteq G$ and V is to construct a Jerrum branching for the group G using some initial ordering of the variables. This provides the following:

- The labels of the branching are a strong generating set for G.
- By creating a branching/forest, rather than a dag, transitivity among the corresponding constraints is maximized. This is explained in the next example.
- The pruning power of the resulting permutations can be estimated using the cost model developed in the previous subsection.

The following example illustrates the second point.

Example 7.7. Consider the generating set $T = \{(1,3),(1,2)(3,4)\}$ as before, with $V = x_4, x_1, x_2, x_3$. $P(T,V)$ thus consists of

$$(x_1 \leq x_3)(x_4 \leq x_3)(x_4 = x_3 \implies x_1 \leq x_2)$$

In this case, $|P(T,V)| = 9$, despite V and T forming a base and SGS, because of the lack of transitivity among the first two constraints. Note that the permutations in T do not describe a branching, since they would correspond to edges from 1 and 4 to 3. By applying the algorithm to construct a Jerrum branching, we obtain the permutations $T' = \{(1,3),(1,2,3,4)\}$ leading to these constraints for $P(T',V)$:

$$(x_1 \leq x_3)(x_4 \leq x_1)(x_4 = x_1 \implies x_1 \leq x_2)((x_4 = x_1)(x_1 = x_2) \implies x_2 \leq x_3)$$

where $|P(T',V)| = 7$. While both T and T' form an SGS with base V, the transitivity among the constraints in $P(T',V)$ allows them to prune a larger portion of $\mathbb{B}^n$.

Implementation Issues

Since Jerrum's algorithm requires $\mathcal{O}(n^5)$ time, it is cannot be used for large values of n. Fortunately, circuit symmetries and symmetries of SAT formulas usually have a block-like structure. In other words, any permutation $g \in G$ can be described by the product $g = h_1 h_2 \ldots h_k$ for $h_i \in H_i$ where each group H_i acts on a different set of points. Therefore, we simply apply Jerrum's algorithm for each H_i. For simplicity, the remainder of the presentation assumes a single branching for G.

7.4.2 Base Change

Having constructed a Jerrum branching with labels T and variable ordering V, we can compute an upper bound on $P(T,V)$ using the cost model developed previously. If R is the set of root nodes in the branching, then

$$|P(T,V)| \leq \Pi_{r \in R} p(r)$$

The structure of the branching determines the right-hand side of the above inequality, and the choice of V determines the structure of the Jerrum branching. Thus, our next task is to determine a good ordering V.

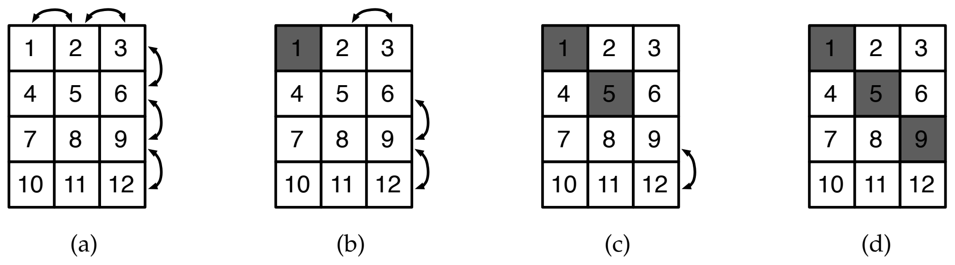


Figure 7.4: Stabilizer chain with base $1, 5, 9$

Recall that a base B is called reduced if each group in the stabilizer chain relative to B is a strict subset of its predecessor. Revisiting the pigeonhole problem in Example 7.4, we find the symmetries of the CNF formula to be

$$
\begin{aligned}
G = \langle \{ & \\
& \left.\begin{array}{l} (1,4)(2,5)(3,6), \\ (4,7)(5,8)(6,9), \\ (7,10)(8,11)(9,12), \end{array}\right\} \text{swap variables for pigeons/rows} \\
& \left.\begin{array}{l} (1,2)(4,5)(7,8)(10,11), \\ (2,3)(5,6)(8,9)(11,12) \end{array}\right\} \text{swap variables for holes/columns} \\
\} \rangle &
\end{aligned}
$$

where the same group can be viewed graphically as the various permutations of rows and columns of the grid in Figure 7.4a. The stabilizer of 1 in G is

$$G_{\{1\}} = \langle \{ (4,7)(5,8)(6,9), (7,10)(8,11)(9,12), (2,3)(5,6)(8,9)(11,12) \} \rangle$$

as shown in Figure 7.4b. The series of diagrams in Figure 7.4 illustrate the stabilizer chain for G using base $B = 1, 5, 9$. This sequence is a reduced base since

$$G \supset G_{\{1\}} \supset G_{\{1,5\}} \supset G_{\{1,5,9\}} = \{()\}$$

The base $1, 5, 6, 9$ would not be reduced because $G_{\{1,5\}} = G_{\{1,5,6\}}$. This fact can be readily seen from Figure 7.4c, as square 6 may only map to itself. Another reduced base for G is the sequence $1, 2, 5, 9$, shown graphically in Figure 7.5. We stress that this base is not *minimal* (since $1, 5, 9$ is a base), but is *reduced* since the following holds:

$$G \supset G_{\{1\}} \supset G_{\{1,2\}} \supset G_{\{1,2,5\}} \supset G_{\{1,2,5,9\}} = \{()\}$$

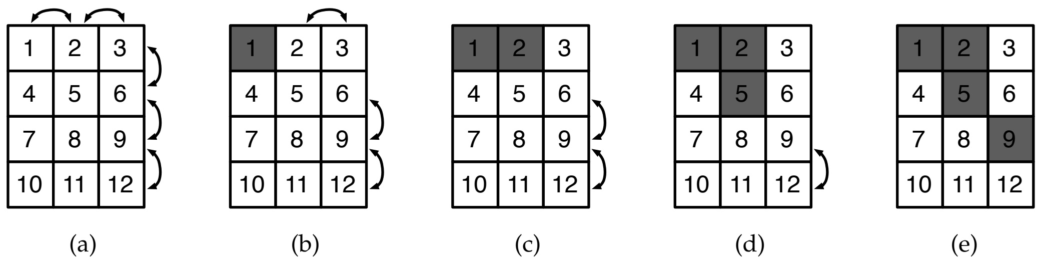


Figure 7.5: Stabilizer chain with base 1, 2, 5, 9

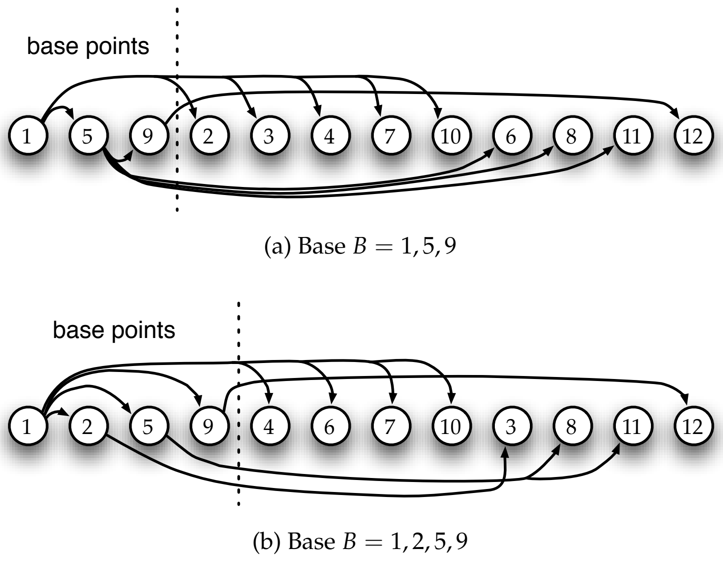


Figure 7.6: Jerrum branchings for two bases

Each of these two bases leads to a different Jerrum branching, shown in Figure 7.6 with labels omitted. Using Proposition 1, we can calculate upper bounds for SBPs generated from the respective labels and variable orders. The upper bound obtained for base 1, 5, 9 is 801 minterms, shown in Figure 7.3, while the upper bound for base 1, 2, 5, 9 is 721 minterms. We have thus shown that a judicious choice for the base can lead to better SBPs.

Using Proposition 1 for a first-order approximation, we seek to minimize the number of leaf nodes in our Jerrum branching as a heuristic to obtain good SBPs. In other words, we want to maximize the number of non-leaf nodes. Each non-leaf node corresponds to a point in a reduced base, so this heuristic is equivalent to finding a maximal-length reduced base.

A simple greedy heuristic for finding a short base is described in [Bla92]—we repeatedly pick β_i from a largest orbit in $G^{(i-1)}$. Then to find a *long* base, we reverse the heuristic and repeatedly pick β_i from a *smallest* nontrivial orbit in $G^{(i-1)}$. Each modification to the Jerrum branching can be performed in $\mathcal{O}(n^3)$ time using the algorithm from [BFP89b].

Example 7.8. Assuming the same group G as in the previous pigeonhole example, we calculate the orbits of G to find the first base point. All points are in the same orbit, so we choose 1 for the first base point. Next, we calculate the orbits of $G_{\{1\}}$ to determine the second base point. As can be seen in Figure 7.5b, the orbits of $G_{\{1\}}$ are $\{\{1\}, \{2,3\}, \{4,7,10\}, \{5,8,11\}, \{6,9,12\}\}$. Thus, we choose 2 (or 3) for the second base point. We continue in this manner until we arrive at the trivial group.

7.4.3 Permutation Simplification

Remark. In this and the following subsection, we assume the natural ordering $V = x_1, x_2, \ldots, x_n$ to remove a layer of indirection and to match the convention in [Jer86]. In the case that V is not the natural ordering, we rewrite T (and all other references to variables in the SAT problem) so that it is. For example, $T = \{(3,4), (1,2,5)\}$ with $V = x_5, x_1, x_2, x_3, x_4$ becomes $T' = \{(4,5), (2,3,1)\}$ and $V' = x_1, x_2, x_3, x_4, x_5$. Formally, supposing ν maps i to v_i, we obtain conjugate permutations $T' = \{\nu\pi\nu^{-1} : \pi \in T\}$.

In subsection 7.3.2, we established that for a given permutation $\pi \in G$, if $k^\pi \neq k$, then the predicate $P(\pi, V)$ contains the following constraint

$$\bigwedge_{i=1}^{k-1} (x_i = x_i^\pi) \implies x_k \leq x_k^\pi$$

whose pruning power depends on the number of points $i < k$ which are moved by π—the more points which are moved, the less powerful the constraint.

Thus, we seek to reduce the number of moved points for π. Here we try to "undo" the movement of k by π. Suppose that $l = k^\pi$ and there exists a permutation $\sigma_{k,l} \in G$ such that $k^{\sigma_{k,l}} = l$. The product $\pi\sigma_{k,l}^{-1}$ is also in group G, and the movement of k is undone, that is $k^{\pi\sigma_{k,l}^{-1}} = k$. Furthermore, suppose $i^{\sigma_{k,l}} = i$ for all $i < k$. Then $P(\sigma_{k,l}, V)$ will already contain the constraint $x_k \leq x_l$, making the above $(\ldots \implies x_k \leq x_l)$ redundant.

The algorithm to simplify a permutation π is shown in Algorithm 9.

- We attempt to undo moves that correspond to paths in the Jerrum branching, because it is easy to identify a permutation π' where $l^{\pi'} = k$. Furthermore, the first moved point of π' is guaranteed to be no smaller than i.
- The algorithm maintains the invariant that i is the first moved point of π, and $i^{\pi} = j$. This implies that we can process the edges of a Jerrum branching, and the results will also form a valid branching.
- The permutation π' on lines 9 and 19 is assumed to have been processed by SIMPLIFY-PERM.
- The check on line 22 is needed to guarantee termination.

We apply SIMPLIFY-PERM on the labels of a branching in topological order from the leaves (Algorithm 10). If SIMPLIFY-PERM results in a permutation which moves fewer points, we keep the result.

Example 7.9. Consider the group G generated by $T = \{(1,3)(2,4,5), (2,4,5), (2,5,4)\}$, shown in Figure 2.2. Let $\pi = (1,3)(2,4,5)$. The procedure SIMPLIFY-PERM(π, G) first tries to reverse any movement on 2, for which $\pi' = (2,4,5)^{-1} = (2,5,4)$. Since $3^{\pi'} = 3$ and $1^{\pi\pi'} = 3$, we replace π by the simpler $\pi\pi' = (1,3)$.

7.4.4 Variable Reordering

In subsection 7.4.2, we maximized the estimated pruning power of the generators T of a group G by finding a base of maximal length. As a side effect, the base points occur as the initial points in the ordering V, meaning that most of the pruning by $P(T, V)$ is concentrated on the variables corresponding to the base points.

The general idea to improve the ordering is as follows. Suppose we have two permutations π and π', and $k^{\pi'} = k$. Then we can move k up in the ordering V, resulting in a V' for which $P(\pi', V) = P(\pi', V')$. However, it is likely that there is less overlap in the portions of $\mathbb{B}^n$ which satisfy the constraints:

$$|P(\pi, V') \cap P(\pi', V')| < |P(\pi, V) \cap P(\pi', V)|$$

that is,

$$|P(\{\pi, \pi'\}, V')| < |P(\{\pi, \pi'\}, V)|$$

Algorithm 9 SIMPLIFY-PERM(π, G)

```
1: $i \leftarrow$ first moved point of $\pi$
2: $j \leftarrow i^{\pi}$
3: for $k \leftarrow i+1$ to $n$ do
4:     $l \leftarrow k^{\pi}$ { note that both $l \geq i$ and $k \geq i$ }
5:     if $k = l$ or $k = j$ and $l = i$ then
6:         skip, nothing to undo
7:     else if $k < l$ and there exists a path from $k$ to $l$ in $G$ then
8:         if there exists an edge $\sigma_{k,l}$ then
9:             $\pi' \leftarrow \sigma_{k,l}^{-1}$
10:        else
11:            $\pi' \leftarrow \tau_l^{-1}\tau_k$
12:            $\pi' \leftarrow$ SIMPLIFY-PERM$(\pi', G)$
13:        end if
14:        if $j^{\pi'} = j$ then
15:            $\pi \leftarrow \pi\pi'$ { now $k^{\pi} = k$ }
16:        end if
17:    else if $k > l$ and there exists a path from $l$ to $k$ then
18:        if there exists an edge $\sigma_{l,k}$ then
19:            $\pi' \leftarrow \sigma_{l,k}$
20:        else
21:            $\pi' \leftarrow \tau_l^{-1}\tau_k$
22:            if $l > i$ then
23:                $\pi' \leftarrow$ SIMPLIFY-PERM$(\pi', G)$
24:            end if
25:        end if
26:        if $j^{\pi'} = j$ then
27:            $\pi \leftarrow \pi\pi'$ { now $k^{\pi} = k$ }
28:        end if
29:    end if
30: end for{ $i$ is still the first moved point of $\pi$, and $i^{\pi}$ is still $j$ }
31: return $\pi$
```

Algorithm 10 SIMPLIFY-PERMS(G)

```
1: for $i \leftarrow n$ downto 1 do
2:     for each edge $\sigma_{i,j}$ do
3:         $\pi \leftarrow$ SIMPLIFY-PERM$(\sigma_{i,j}, G)$
4:         if $\pi$ moves fewer points than $\sigma_{i,j}$ then
5:             $\sigma_{i,j} \leftarrow \pi$
6:             update $\tau$
7:         end if
8:     end for
9: end for
```

Example 7.10. Suppose $T = \{(1,4)(2,5)(3,6)(7,8), (2,6), (3,5)\}$ and $V = x_1, x_2, \ldots, x_8$. The pruning power of this combination is $|P(T,V)| = 96$. Little pruning is performed on x_7 and x_8 because they occur last in the ordering V. Some improvement can be attained by moving 7 and 8 up in the ordering—for $V' = x_1, x_7, x_8, x_2, x_3, x_4, x_5, x_6$, $|P(T,V')| = 86$. However, we can do better using $V'' = x_1, x_4, x_7, x_8, x_2, x_6, x_3, x_5$, for which $|P(T,V'')| = 78$.

The algorithm to find a new ordering is described in Algorithm 11. For any k which is not a base point, BUBBLE-POINTS finds the smallest i such that $G^{(i)}$ stabilizes k. The permutations T remain an SGS for $\langle T \rangle$ as long as k appears after i—BUBBLE-POINTS puts k after i and before the next point in the reduced base (if any).

Algorithm 11 BUBBLE-POINTS(T)

```
1:  for $k \leftarrow 1$ to $n$ do
2:      $s_k \leftarrow 0$
3:  end for
4:  for each $\pi \in T$ do
5:      $i \leftarrow$ first moved point of $\pi$
6:      for $k \leftarrow 1$ to $n$ do
7:          if $k^\pi \neq k$ and $s_k < i$ then
8:              $s_k \leftarrow i$
9:          end if
10:     end for
11: end for
12: $V \leftarrow 1, 2, \ldots, n$
13: stable-sort $V$ using $s$ as keys
14: return $V$
```

Example 7.11. Let T be the same as in the previous example. BUBBLE-POINTS(T) computes $s = 1,2,3,1,3,2,1,1$ and returns $V = 1,4,7,8,2,6,3,5$.

7.5 Experimental Results

To test the effective of our proposed algorithms, we examined an assortment of SAT benchmarks from synthesis and verification:

- ENGINE-UNSAT v1.0, PIPE-UNSAT v1.1, VLIW-UNSAT v2.0 from `http://www.miroslav-velev.com/sat_benchmarks.html`. These examples all come from microprocessor verification.

- DAC'02 benchmarks from http://www.aloul.net/benchmarks.html, which are used in [ARSM03]. These consist of instances of pigeonhole problems, Urquhart problems [Urq87], and instances from FPGA synthesis.

All experiments are performed on an Intel Xeon E5345 2.33GHz processor, and all runtimes are reported in milliseconds. SAT formulas are solved using Minisat version 1.14[3]. We use as a baseline the approach described in [ASM06], which takes symmetry generators from Saucy [DLSM04] and directly translates them into SBPs. Therefore, we do not report the time required to find symmetries, nor do we report the time to solve any of the SAT instances without SBPs.

Upon further inspection, we discovered that for the microprocessor verification instances, all orbits are of size one or two. For example, the symmetries of one instance may look like:

$$\begin{aligned} T &= \{(1,2)(3,4)(5,6),(7,8),(9,10)(11,12)\} \\ G &= \langle T \rangle \end{aligned}$$

where x_1 can only move to x_2, x_3 can only move to x_4, etc. Note that T already forms a strong generating set for G for any base. For all intents and purposes, T cannot be further simplified nor strengthened. Thus, in our experiments, we focus on the DAC'02 suite of SAT instances.

Given a SAT instance ϕ, we convert it into a graph and obtain symmetry generators from Saucy. From these generators, we evaluate six combinations of algorithms to derive a set of SBPs $P(T,V)$:

- "Baseline" — T consists of the symmetry generators, and $V = x_1, x_2, \ldots, x_n$.
- 4 (i.e., subsection 7.4.4) — T consists of the symmetry generators, and V is determined by BUBBLE-POINTS.
- 1 (i.e., subsection 7.4.1) — T consists of labels from a Jerrum branching, and $V = x_1, x_2, \ldots, x_n$.
- 1,3 — T consists of labels from a Jerrum branching processed by SIMPLIFY-PERMS, and $V = x_1, x_2, \ldots, x_n$.

[3] http://minisat.se/MiniSat.html

- 1,3,4 — T consists of labels from a Jerrum branching processed by SIMPLIFY-PERMS, and V is determined by BUBBLE-POINTS.
- 1,2,3,4 — We find a maximal-length base for the symmetry group. T consists of labels from a Jerrum branching processed by SIMPLIFY-PERMS, and V is determined by BUBBLE-POINTS.

The results for the DAC'02 instances are shown in Table 7.1 as six pairs of columns. The first number in each pair is the time to derive SBPs from symmetry generators, and the second number is the time required by Minisat to solve $\phi \wedge P(T, V)$. A dash indicates that Minisat is unable to solve the instance within 900 seconds. Furthermore the benchmarks in Table 7.1 are divided into two sets: the instances in the top half are unsatisfiable, while the instances in the bottom half are satisfiable. From these results, we can make a few observations:

- Besides the Urquhart problems, every instance is easily solved using the baseline SBPs.
- The Urquhart problems are difficult using the baseline SBPs, but trivial once we compute a strong generating set.
- Finding a maximal-length base (subsection 7.4.2) greatly increases the time to generate SBPs.

In the next set of experiments, we take the same set of SAT instances, and create 25 variants of each in which the variables are randomly renumbered. We then provide these instances to Saucy and again use the same six combinations of algorithms to derive SBPs. The results are shown in Table 7.2—except for Urq5_5, the reported runtimes are averages over the 25 variants. These results show:

- reordering the variables in V using BUBBLE-POINTS yields stronger SBPs on the chnl and hole series of benchmarks, even without finding a strong generating set for T.
- on the same series of benchmarks, SIMPLIFY-PERMS and BUBBLE-POINTS greatly reduce the time required by Minisat (columns "1,3" and "1,3,4").
- Base-length maximization as suggested in subsection 7.4.2 does not greatly affect the SAT solution time (columns "1,2,3,4").

Benchmark	Baseline		SBP Strengthening									
			4		1		1,3		1,3,4		1,2,3,4	
Urq3_5	0	140	0	132	8	0	8	0	8	4	4	0
Urq4_5	4	1400	4	1380	4	4	16	0	8	0	16	0
Urq5_5	4	311623	8	311891	24	4	28	4	28	4	56	4
Urq6_5	4	—	8	—	68	0	76	0	76	0	160	4
Urq7_5	8	—	12	—	156	0	156	8	168	4	344	0
Urq8_5	12	—	16	—	356	4	364	0	364	0	804	0
chnl10_11	0	0	0	0	112	32	136	16	132	20	316	16
chnl10_12	4	4	4	0	136	32	176	24	180	16	408	16
chnl10_13	4	0	8	8	164	32	208	20	220	24	484	20
chnl11_12	4	4	8	4	172	44	216	24	220	24	508	20
chnl11_13	4	4	12	4	200	56	272	28	276	24	624	32
chnl11_20	8	4	8	4	564	104	916	80	924	72	2052	64
hole7	8	0	0	4	8	4	4	4	8	0	8	4
hole8	0	4	0	0	8	4	12	4	12	4	12	0
hole9	0	0	0	0	20	12	24	8	20	0	32	4
hole10	0	8	0	0	24	16	32	8	44	8	44	12
hole11	0	0	0	4	40	24	60	16	56	12	76	12
hole12	0	0	8	0	56	24	92	20	92	16	120	24
fpga10_8	0	0	0	0	28	8	28	12	32	16	32	4
fpga10_9	0	4	4	0	20	4	24	4	20	4	28	0
fpga12_8	0	4	4	0	36	20	32	20	36	16	44	4
fpga12_9	0	4	0	8	36	12	28	8	32	8	44	8
fpga12_11	4	8	4	8	44	20	52	12	52	12	72	4
fpga12_12	0	0	0	8	92	28	96	36	96	28	120	20
fpga13_9	4	4	0	4	40	16	36	12	36	16	56	4
fpga13_10	4	4	4	8	48	16	44	12	48	12	68	4
fpga13_12	4	4	4	4	76	28	72	12	72	16	116	4
s3-3-3-1	8	28	8	32	120	192	120	188	128	188	176	240
s3-3-3-3	12	60	8	60	232	120	232	112	236	120	316	220
s3-3-3-4	12	52	4	52	144	204	140	208	152	200	220	40
s3-3-3-8	4	28	16	24	184	140	180	144	196	136	268	296
s3-3-3-10	8	124	12	128	228	228	224	228	244	228	296	120

Table 7.1: DAC'02 benchmarks with varying methods of symmetry breaking

- Overall, the combination "1,3,4" performs best over the entire set of SAT problems.

In summary, the baseline method of producing SBPs works well on every instance in the DAC'02 benchmarks, with the exception of the Urq series. When the variables are randomly renamed/reordered, Saucy produces poor symmetry generators for use with the baseline method. However, applying our proposed algorithms allows us to construct a useful set of SBPs.

7.6 Conclusion

In this chapter we explored various issues that arise when generating so-called "symmetry breaking predicates" for Boolean satisfiability problems. We adapt the concepts of stabilizer chains and strong generating sets from computational group theory to analyze properties of potential predicates. These concepts enable efficient heuristics for formulating effective predicates. Experimental results show that our reformulated predicates allow significant and consistent speedups compared to previous work.

Benchmark	Baseline		SBP Strengthening									
			4		1		1,3		1,3,4		1,2,3,4	
Urq3_5	2	108	2	107	3	1	4	1	4	1	6	1
Urq4_5	2	4586	3	4616	8	1	9	1	9	1	16	1
Urq5_5†	4	264581	3	265015	27	1	26	1	26	1	56	2
Urq6_5	5	—	5	—	75	1	76	2	76	1	165	1
Urq7_5	6	—	7	—	155	2	156	2	156	2	340	1
Urq8_5	10	—	11	—	360	3	363	3	364	3	808	2
chnl10_11	5	299	5	66	129	67	132	42	134	11	303	21
chnl10_12	4	308	6	63	155	71	161	47	163	10	369	21
chnl10_13	5	295	6	62	191	72	199	46	200	12	452	22
chnl11_12	5	1594	5	301	197	184	203	131	208	13	472	42
chnl11_13	5	2013	7	264	238	183	248	115	250	14	568	43
chnl11_20	11	2303	11	331	691	226	725	144	729	24	1664	54
hole7	1	4	1	3	8	3	8	4	8	2	14	3
hole8	2	12	1	5	14	8	14	8	15	3	27	3
hole9	2	49	2	14	23	19	23	15	23	4	43	6
hole10	2	233	2	59	35	56	36	38	35	6	68	14
hole11	3	1641	2	211	52	156	52	105	52	8	102	31
hole12	3	15210	4	1052	78	490	79	282	77	12	151	68
fpga10_8	1	4	2	5	23	13	23	12	23	8	38	8
fpga10_9	3	2	3	5	31	11	30	11	30	8	50	5
fpga12_8	2	5	4	4	35	20	36	22	37	15	57	13
fpga12_9	4	8	3	6	43	40	45	33	47	18	73	18
fpga12_11	4	8	4	6	70	40	70	40	71	17	119	27
fpga12_12	3	5	4	5	80	34	85	104	85	24	152	56
fpga13_9	4	8	3	5	54	57	55	74	56	29	88	27
fpga13_10	3	13	4	12	72	57	74	81	72	29	121	61
fpga13_12	5	10	5	7	111	81	119	95	118	31	193	264
s3-3-3-1	10	768	9	420	86	415	87	318	92	328	144	388
s3-3-3-3	12	641	11	446	124	483	125	371	134	355	206	337
s3-3-3-4	10	317	9	264	97	390	100	428	107	202	166	213
s3-3-3-8	11	594	10	621	108	467	107	493	115	326	179	328
s3-3-3-10	12	751	13	550	131	558	131	379	143	360	217	410

†16 of 25 variants solved using SBP methods “Baseline” and “4”. Nine remaining unsolved variants are not included in the average.

Table 7.2: Randomized DAC’02 benchmarks with varying methods of symmetry breaking

8. Conclusion

In this dissertation, we looked at the use of functional symmetries in the context of circuits and Boolean formulas, where symmetries are permutations of connections/variables which do not change the functionality/meaning. In particular, we explored the possibility of using symmetries to *expand* the solution space for optimization problems. This may help overcome some of the limitations of a linear EDA flow which traps a design into a local minimum at every stage. We also explored the use of symmetries to *shrink* the search space for search problems.

Now we summarize the contributions of this dissertation and point out the strengths and weaknesses of our approach.

8.1 Contributions

In chapter 3, we described an approach to symmetry detection for functions over many variables. We provide a compact reduction from functions to graphs which avoids the exponential size requirement of the naïve formulation. This allows us to exploit recent advances in graph automorphism solvers, and to easily analyze functions which have multiple outputs. In comparison, related work in Boolean matching [CK06, ABPS07] does not work on multiple-output functions.

In chapter 4, we introduced the concept of circuit symmetries as permutations of connections within a circuit. We distinguish between three families of symmetries, each a subgroup of the group $G \cup B \cup I$, and discuss the applicability of each subgroup towards a different problem in CAD.

In chapter 5, we tackled the problem of technology mapping, also known as library binding. The technology mapping problem usually assumes a fixed subject graph of AND and INV nodes. We showed how symmetries can be used to derive alternative decompositions of functions, which may permit better solutions from the technology mapper. For efficiency, our approach divides the group G into cosets. Wherever connections in the subject graph can be arbitrarily permuted, we use a partitioning algorithm to perform restructuring over subgroup $H \leq G$. For other symmetries, we enumerate permutations from the transversal $G : H$. Our results show that symmetries (particularly those of AND trees) allow us to greatly reduce the area required after technology mapping.

In chapter 6, we used symmetries to reduce wirelength during the placement stage of synthesis. Many current placement algorithms are based on iteratively moving cells in decreasing amounts until a final legal placement is reached; the connectivity of the network is assumed to be fixed throughout the placement flow. We use symmetries to remove this restriction, restructuring the network between placement iterations. Similar to in the technology mapping problem, we can restructure the netlist using the full set of symmetries $G \cup B$, or simply those over AND trees. Our results show that usage of symmetries provides a consistent improvement in wirelength over an unmodified version of the Capo placer.

In chapter 7, we addressed the problem of deriving symmetry-breaking predicates for SAT formulas. We showed how directly translating generators from a symmetry group into predicates does not lead to the most effective predicates in terms of minimizing the time required by a SAT solver, and provide algorithms to obtain short and effective predicates for breaking symmetries. Experiments with Minisat show that our algorithms make certain hard SAT problems trivial, and guard the SAT solving process against bad variable orderings.

8.2 Strengths and Weaknesses

A major benefit of using symmetries for expanding the solution space during synthesis is that no additional Boolean reasoning is required after identifying symmetries—all manipulation afterwards is purely structural. For example, area-oriented optimization in chapter

5 is performed using graph partitioning. This implies that optimization using symmetries is potentially more efficient than other methods.

Another benefit of symmetries is that they exist at all stages of the design flow, and can be used at any time, since the concept of connections between components exists at all stages. Furthermore, exchanging connections is much simpler to carry out at later stages of the design flow, since the routing of wires is one of the final stages. In contrast, resynthesis after placement will require updating the placement to accommodate new cells.

The main weakness of our approach is that many of the circuits and SAT problems that we examined do not exhibit a large number of complex symmetries. By this, we mean that in circuits, most of the symmetries we found were those of AND trees—functions such as $ab + c(a + b)$ do not occur very often, or are not found by our procedure. In SAT problems, many of the symmetries are limited to swaps between two sets of variables, that is, there is a one-to-one correspondence between variables in sets X_1 and X_2.

In the case of Boolean satisfiability, this means that symmetry-breaking predicates may not be able to prune much of the search space. For swaps between $X_1 = \{x_1, x_2, x_3\}$ and $X_2 = \{x_4, x_5, x_6\}$, the symmetry-breaking predicate from the permutation $(1, 4)(2, 5)(3, 6)$ cannot prune more than $1/2$ of the search space.

In the case of synthesis problems in which we expand the solution space, i.e. technology mapping and placement, the limited set of symmetries is a mixed blessing. On the one hand, this implies that there is not much potential beyond optimizing AND trees. On the other hand, the most efficient algorithms that we know of are those that operate over symmetries of AND trees.

Bibliography

[ABPS07] Giovanni Agosta, Francesco Bruschi, Gerardo Pelosi, and Donatella Sciuto. A unified approach to canonical form-based Boolean matching. In *Design Automation Conference*, pages 841–846, 2007.

[AP05] Afshin Abdollahi and Massoud Pedram. A new canonical form for fast Boolean matching in logic synthesis and verification. In *Design Automation Conference*, pages 379–384, 2005.

[ARSM03] Fadi A. Aloul, Arathi Ramani, Karem A. Sakallah, and Igor L. Markov. Solving difficult instances of Boolean satisfiability in the presence of symmetry. *IEEE Transactions on CAD*, 22(9):1117–1137, September 2003.

[Ash59] R. Ashenhurst. The decomposition of switching functions. In *International Symposium on the Theory of Switching*, volume 29 of *Annals of the Computation Laboratory of Harvard University*, pages 74–116. Harvard University Press, 1959.

[ASM06] Fadi A. Aloul, Karem A. Sakallah, and Igor L. Markov. Efficient symmetry breaking for Boolean satisfiability. *IEEE Transactions on Computers*, 55(5):549–558, May 2006.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *5th International Conference on*

Tools and Algorithms for Construction and Analysis of Systems, pages 193–207, Amsterdam, The Netherlands, March 1999.

[BCF$^+$91] László Babai, Gene Cooperman, Larry Finkelstein, Eugene M. Luks, and Ákos Seress. Fast Monte Carlo algorithms for permutation groups. In *Annual ACM Symposium on Theory of Computing*, pages 90–100, 1991.

[Ber] Berkeley Logic Synthesis and Verification Group. ABC: A system for sequential synthesis and verification. http://www.eecs.berkeley.edu/~alanmi/abc/.

[BFP89a] Cynthia A. Brown, Larry Finkelstein, and Paul Walton Purdom, Jr. Backtrack searching in the presence of symmetry. In *International Conference on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, Lecture Notes in Computer Science*, volume 357. Springer, 1989.

[BFP89b] Cynthia A. Brown, Larry Finkelstein, and Paul Walton Purdom, Jr. A new base change algorithm for permutation groups. *SIAM Journal on Computing*, 18(5):1037–1047, 1989.

[BHMSV84] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[Bla92] Kenneth D. Blaha. Minimum bases for permutation groups: The greedy approximation. *Journal of Algorithms*, 13(2):297–306, 1992.

[Bre77] Melvin A. Breuer. A class of min-cut placement algorithms. In *Design Automation Conference*, pages 284–290, 1977.

[Cad05] Cadence Design Systems. CRETE 180nm generic library. http://iwls.org/iwls2005/benchmarks.html, 2005.

[CD94] Jason Cong and Yuzheng Ding. FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. *IEEE Transactions on CAD*, 13(1):1–12, January 1994.

[CGLR96] James Crawford, Matthew Ginsberg, Eugue Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *Fifth International Conference on Principles of Knowledge Representation and Reasoning*, 1996.

[Cha07] Satrajit Chatterjee. *On Algorithms for Technology Mapping*. PhD thesis, UC Berkeley, 2007.

[CHH$^+$04] Chih-Wei Chang, Ming-Fu Hsiao, Bo Hu, Kai Wang, Malgorzata Marek-Sadowska, Chung-Kuan Cheng, and Sao-Jie Chen. Fast postplacement optimization using functional symmetries. *IEEE Transactions on CAD*, 23(1):102–118, January 2004.

[CJ99] Malgorzata Chrzanowska-Jeske. Generalized symmetric and generalized pseudo-symmetric functions. In *Electronics, Circuits and Systems, 1999. Proceedings of ICECS '99. The 6th IEEE International Conference on*, 1999.

[CK84] Chung-Kuan Cheng and Ernest S. Kuh. Module placement based on resistive network optimization. *IEEE Transactions on CAD*, 3(3), July 1984.

[CK06] Donald Chai and Andreas Kuehlmann. Building a better Boolean matching and symmetry detector. In *Design Automation and Test in Europe*, pages 1079–1084, 2006.

[CKM00] Andrew E. Caldwell, Andrew B. Kahng, and Igor L. Markov. Can recursive bisection alone produce routable placements? In *Design Automation Conference*, pages 693–698, 2000.

[CLR89] Thomas H. Cormen, Charles E. Leiserson, and Ronald L Rivest. *Introduction to Algorithms*. McGraw-Hill, 1989.

[CMB05a] Kai-Hui Chang, Igor L. Markov, and Valeria Bertacco. Post-placement rewiring and rebuffering by exhaustive search for functional symmetries. In *International Conference on CAD*, pages 56–63, 2005.

[CMB$^+$05b] Satrajit Chatterjee, Alan Mishchenko, Robert K. Brayton, Xinning Wang, and Timothy Kam. Reducing structural bias in technology mapping. In *International Conference on CAD*, pages 519–526, 2005.

[Cor03] Jordi Cortadella. Timing-driven logic bi-decomposition. *IEEE Transactions on CAD*, 22(6):675–685, June 2003.

[Cou97] Olivier Coudert. Gate sizing for constrained delay/power/area optimization. *IEEE Transactions on VLSI*, 5(4):465–472, December 1997.

[CS03] Jovanka Ciric and Carl Sechen. Efficient canonical form for Boolean matching of complex functions in large libraries. *IEEE Transactions on CAD*, 22(5):535–544, May 2003.

[CWD99] Jason Cong, Chang Wu, and Yuzheng Ding. Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution. In *International Symposium on FPGAs*, pages 29–35, 1999.

[DGK94] Srinivas Devadas, Abhijit Ghosh, and Kurt Keutzer. *Logic Synthesis*. McGraw-Hill, 1994.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, July 1962.

[DLSM04] Paul T. Darga, Mark H. Liffiton, Karem A. Sakallah, and Igor L. Markov. Exploiting structure in symmetry detection for CNF. In *Design Automation Conference*, pages 530–534, 2004.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the Association of for Computing Machinery*, 7:102–215, 1960.

[DSM08] Paul T. Darga, Karem A. Sakallah, and Igor L. Markov. Faster symmetry discovery using sparsity of symmetries. In *Design Automation Conference*, pages 149–154, 2008.

[EH78] Colin R. Edwards and S. L. Hurst. A digital synthesis procedure under function symmetries and mapping methods. *IEEE Transactions on Computers*, 27(11):985–997, 1978.

[ES03] Niklas Eén and Niklas Sörensson. An extensible SAT solver. In *Theory and Applications of Satisfiability Testing*, 2003.

[GAP06] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.4*, 2006. http://www.gap-system.org.

[Hak85] A. Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985.

[Hal59] Marshall Hall. *The Theory of Groups*. Macmillan, 1959.

[HK98] Uwe Hinsberger and Reiner Kolla. Boolean matching for large libraries. In *Design Automation Conference*, 1998.

[IWL05] IWLS 2005 benchmarks. http://iwls.org/iwls2005/benchmarks.html, 2005.

[Jer86] Mark Jerrum. A compact representation for permutation groups. *Journal of Algorithms*, 7(1):60–78, 1986.

[JKCMS97] Yi-Min Jiang, Angela Krstic, Kwang-Ting Cheng, and Malgorzata Marek-Sadowska. Post-layout logic restructuring for performance optimization. In *Design Automation Conference*, pages 662–665, 1997.

[KD91] Bo-Gwan Kim and Donald L. Dietmeyer. Multilevel logic synthesis of symmetric switching functions. *IEEE Transactions on CAD*, 10(4):436–446, April 1991.

[Keu87] Kurt Keutzer. DAGON: Technology binding and local optimization by DAG matching. In *Design Automation Conference*, pages 341–347, 1987.

[Kis98] Andrzej Kisielewicz. Symmetry groups of Boolean functions and constructions of permutation groups. *Journal of Algebra*, 199:379–403, 1998.

[KK08] Neil Kettle and Andy King. An anytime algorithm for generalized symmetry detection in robdds. *IEEE Transactions on CAD*, 27(4):764–777, April 2008.

[KL70] Brian Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *Bell Systems Technical Journal*, 49(1):291–307, 1970.

[Knu91] Donald E. Knuth. Efficient representation of perm groups. *Combinatorica*, 11(1):33–43, 1991.

[KP08] Volker Kaibel and Marc E. Pfetsch. Packing and partitioning orbitopes. *Mathematical Programming*, 114(1):1–36, 2008.

[KR89] Kurt Keutzer and D. Richards. Computational complexity of logic synthesis and optimization. In *International Workshop on Logic Synthesis*, 1989.

[Kri85] Balakrishnan Krishnamurthy. Short proofs for tricky formulas. *Acta Informatica*, 22(3):253–275, 1985.

[KS00] Victor N. Kravets and Karem Sakallah. Constructive library-aware synthesis using symmetries. In *Design Automation and Test in Europe*, 2000.

[KSJA91] J. M. Kleinhans, G. Sigl, F. M. Johannes, and K. J. Antreich. GORDIAN: VLSI placement by quadratic programming and slicing optimization. *IEEE Transactions on CAD*, 10(3):356–365, March 1991.

[Leo91] Jeffrey S. Leon. Permutation group algorithms based on partitions, I: Theory and algorithms. *Journal of Symbolic Computation*, 12:533–583, 1991.

[LESJ98] Aiguo Lu, Hans Eisenmann, Guenter Stenz, and Frank M. Johannes. Combining technology mapping with post-placement resynthesis for performance optimization. In *International Conference on Computer Design*, pages 616–621, 1998.

[Lev74] Giorgio Levi. Graph isomorphism: A heuristic edge-partitioning-oriented algorithm. *Computing*, 12(4):291–313, December 1974.

[LJL08] Hsuan-Po Lin, Jie-Hong Roland Jiang, and Ruei-Rung Lee. To SAT or not to SAT: Ashenhurst decomposition in a large scale. In *International Conference on CAD*, 2008.

[LR02] Eugene M. Luks and Amitabha Roy. Symmetry breaking in constraint satisfaction. In *Intl. Conf. of Artificial Intelligence and Mathematics*, 2002.

[Luk99] Eugene M. Luks. Hypergraph isomorphism and structural equivalence of Boolean functions. In *ACM Symposium on Theory of Computing*, pages 652–658, 1999.

[LWGH97] Eric Lehman, Yosinoro Watanabe, Joel Grodstein, and Heather Harkness. Logic decomposition during technology mapping. *IEEE Transactions on CAD*, 16(8):813–834, August 1997.

[Mau06] Peter M. Maurer. Using conjugate symmetries to enhance gate-level simulations. In *Design Automation and Test in Europe*, 2006.

[MBC08] Alan Mishchenko, Robert K. Brayton, and Satrajit Chatterjee. Boolean factoring and decomposition of logic networks. In *International Conference on CAD*, 2008.

[MCB07] Alan Mishchenko, Satrajit Chatterjee, and Robert K. Brayton. Improvements to technology mapping for LUT-based FPGAs. *IEEE Transactions on CAD*, 26(2):240–253, February 2007.

[McC56] Edward J. McCluskey. Group invariance or total symmetry. *Bell Systems Technical Journal*, 35(6):1445–1453, November 1956.

[McK81] Brendan D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[Mis03] Alan Mishchenko. Fast computation of symmetries in Boolean functions. *IEEE Transactions on CAD*, 22(11):1588–1593, November 2003.

[Miy82] Musashi Miyamoto. *The Book of Five Rings*. Bantam, 1982.

[MM90] Frédéric Mailhot and Giovanni De Micheli. Technology mapping using Boolean matching and don't care sets. In *European Design Automation Conference*, pages 212–216, 1990.

[MMM95] Janett Mohnke, Paul Molitor, and Sharad Malik. Limits of using signatures for permutation independent Boolean comparison. In *Asia and South Pacific Design Automation Conference*, pages 459–464, 1995.

[MMZ$^+$01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535, 2001.

[MSS99] João P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on CAD*, 48:506–521, 1999.

[Mun57] James Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, March 1957.

[Mur71] Saburo Muroga. *Threshold Logic And its Applications*. John Wiley, 1971.

[Nam06] Gi-Joon Nam. ISPD06 placement contest and benchmark suite. In *International Symposium on Physical Design*, 2006.

[Nan08] Nangate Inc. Nangate 45nm open cell library. http://www.nangate.com/openlibrary/, 2008.

[NSR99] Gi-Joon Nam, Karem A. Sakallah, and Rob A. Rutenbar. Satisfiability-based detailed FPGA routing. In *International Conference On VLSI Design*, pages 574–577, 1999.

[Pet08] Graham Petley. vsclib standard cell library. http://vlsitechnology.org/html/vsc_description.html, 2008.

[PR94] Irith Pomeranz and Sudhakar M. Reddy. On determining symmetries in inputs of logic circuits. *IEEE Transactions on CAD*, 13(11):1428–1434, November 1994.

[Roy07] Amitabha Roy. Symmetry-breaking formulas for groups with bounded orbit projections. In *International Workshop on Symmetry in Constraint Satisfaction Problems*, 2007.

[Rud89] Richard L. Rudell. *Logic synthesis for VLSI design*. PhD thesis, UC Berkeley, 1989.

[SBSV96] Paul R. Stephan, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Combinational test generation using Boolean satisfiability. *IEEE Transactions on CAD*, 15(9):1167–1176, September 1996.

[Sec88] Carl Sechen. *VLSI placement and global routing using simulated annealing*. Kluwer Academic Publishers, 1988.

[Ser02] Ákos Seress. *Permutation Group Algorithms*, volume 152 of *Cambridge Tracts in Mathematics*. Cambridge University Press, 2002.

[Sim71] Charles C. Sims. Computation with permutation groups. In *Symposium on Symbolic and Algebraic Manipulation*, pages 23–28, 1971.

[SS77] Richard M. Stallman and Gerald Jay Sussman. Forward reasoning and dependency directed backtracking in a system for computer aided circuit analysis. *Artificial Intelligence*, 9(2):135–196, 1977.

[SVBY06] Sean Safarpour, Andreas Veneris, Gregg Baeckler, and Richard Yuan. Efficient SAT-based boolean matching for FPGA technology mapping. In *Design Automation Conference*, pages 466–471, 2006.

[Urq87] Alasdair Urquhart. Hard examples for resolution. *Journal of the ACM*, 34(1):209–219, 1987.

[VKBSV97] Tiziano Villa, Timothy Kam, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Explicit and implicit algorithms for binate covering problems. *IEEE Transactions on CAD*, 16(7):677–691, July 1997.

[Wan06] Kuo-Hua Wang. Exploiting k-distance signature for Boolean matching and G-symmetry detection. In *Design Automation Conference*, pages 516–521, 2006.

[WD98] Feng Wang and Donald L. Dietmeyer. Exploiting near symmetry in multilevel logic synthesis. *IEEE Transactions on CAD*, 17(9):772–781, September 1998.

[WKSV03] G. Wang, A. Kuehlmann, and A. Sangiovanni-Vincentelli. Structural detection of symmetries in Boolean functions. In *International Conference on Computer Design*, pages 498–503, 2003.

[YM91] Jerry Chih-Yuan Yang and Giovanni De Micheli. Spectral techniques for technology mapping. Technical Report CSL-TR-91-498, Stanford University, 1991.

[ZMMM01] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *International Conference on CAD*, pages 279–285, 2001.

Index