[SG95] Barbara M. Smith and Stuart A. Grant. Sparse constraint graphs and exceptionally hard problems. In *Proceedings of IJCAI-95: 14th International Joint Conference on Artificial Intelligence*, volume 1, pages 646–654, Montreal, Canada, August 1995.

[SG97] Barbara M. Smith and Stuart A. Grant. Modelling exceptionally hard constraint satisfaction problems. In *CP-97: 3rd International Conference on Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 182–195, Austria, October 1997.

[SK96] B. Selman and S. Kirkpatrick. Finite-Size Scaling of the Computational Cost of Systematic Search. *Artificial Intelligence*, 81(1–2):273–295, 1996.

[SKC96] Bart Selman, Henry Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring and Satisfiability: the Second DIMACS Implementation Challenge*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 521–532. American Mathematical Society, 1996.

[ST94] G. Samorodnitsky and M. Taqqu. *Stable Non-Gaussian Random Processes: Stochastic Models with Infinite Variance.* Chapman and Hall, 1994.

[SW01] John K. Slaney and Toby Walsh. Backbones in optimization and approximation. In *Proceedings of IJCAI-01: 17th International Joint Conference on Artificial Intelligence*, pages 254–259, Seattle, WA, August 2001.

[Sze05] S. Szeider. Backdoor sets for DLL subsolvers. *Journal of Automated Reasoning*, 35(1-3):73–88, 2005. Special issue on SAT 2005.

[Tri96] Michael Trick. `http://mat.gsia.cmu.edu/color/solvers/trick.c`. Source code for the implementation of an exact deterministic algorithm for graph coloring based on DSATUR, 1996.

[Wal99] T. Walsh. Search in a small world. In *Proceedings of IJCAI-99: 16th International Joint Conference on Artificial Intelligence*, 1999.

[WGS03a] R. Williams, C. P. Gomes, and B. Selman. Backdoors to typical case complexity. In *Proceedings of IJCAI-03: 18th International Joint Conference on Artificial Intelligence*, 2003.

[WGS03b] R. Williams, C. P. Gomes, and B. Selman. On the connections between backdoors, restarts, and heavy-tailedness in combinatorial search. In *Proceedings of SAT-03: 6th International Conference on Theory and Applications of Satisfiability Testing*, 2003.

[Wol02] Stephen Wolfram. *A New Kind of Science.* Stephen Wolfram, 2002.

[Zha02] H. Zhang. A random jump strategy for combinatorial search. In *International Symposium on AI and Math*, Fort Lauderdale, FL, 2002.

Handbook of Satisfiability
Armin Biere, Marijn Heule, Hans van Maaren and Toby Walsh (Eds.)
IOS Press, 2009

doi:10.3233/978-1-58603-929-5-289

Chapter 10

Symmetry and Satisfiability

Karem A. Sakallah

Symmetry is at once a familiar concept (we recognize it when we see it!) and a profoundly deep mathematical subject. At its most basic, a symmetry is some transformation of an object that leaves the object (or some aspect of the object) unchanged. For example, Fig. 10.1 shows that a square can be transformed in eight different ways that leave it looking exactly the same: the identity do-nothing?transformation, 3 rotations, and 4 mirror images (or reflections). In the context of decision problems, the presence of symmetries in a problem's search space can frustrate the hunt for a solution by forcing a search algorithm to fruitlessly explore symmetric subspaces that do not contain solutions. Recognizing that such symmetries exist, we can direct a search algorithm to look for solutions only in non-symmetric parts of the search space. In many cases, this can lead to significant pruning of the search space and yield solutions to problems which are otherwise intractable.

In this chapter we will be concerned with the symmetries of Boolean functions, particularly the symmetries of their conjunctive normal form (CNF) representations. Our goal is to understand what those symmetries are, how to model them using the mathematical language of group theory, how to derive them from a CNF formula, and how to utilize them to speed up CNF SAT solvers.

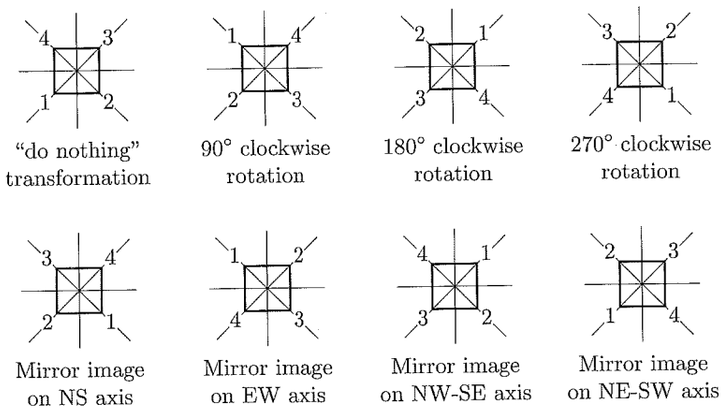


Figure 10.1. Symmetries of the square

10.1. Motivating Example

Consider the Boolean function $f(a, b, c)$ specified in Table 10.1. It can be expressed algebraically in sum-of-minterms form as

$$f(a, b, c) = a'bc + ab'c \tag{10.1}$$

It is not too difficult to observe that this function remains invariant under a swap of inputs a and b:

$$f(b, a, c) = b'ac + ba'c = ab'c + a'bc = a'bc + ab'c \tag{10.2}$$

It is also rather obvious that this invariance follows from the commutativity of the Boolean AND and OR operators. Viewing this function as an "object" with three "features" labeled a, b, and c, we may consider this swap as a transformation that leaves the essential nature of the object (its "function") unchanged, hence a symmetry of the object. Allowing for possible inversion, in addition to swaps, of the inputs we can readily check that the following two transformations are also symmetries of this function:

$$f(b', a', c) = b''a'c + b'a''c = a'bc + ab'c \tag{10.3}$$
$$f(a', b', c) = a''b'c + a'b''c = ab'c + a'bc = a'bc + ab'c \tag{10.4}$$

Besides commutativity, invariance under these two transformations relies on the involution of double complementation, i.e., the fact that $x'' = x$. Note that the transformation in (10.3) can be described as a swap between a and b' (equivalently between b and a'). On the other hand, the transformation in (10.4) is a *simultaneous swap* between a and a', and between b and b'.

To facilitate later discussion, let us label these swaps as follows:

Label	Swap
ε	No swap
α	Swap a with b
β	Swap a with b'
γ	Swap a with a' and b with b'

A natural question now arises: are these the only possible transformations that leave this function unchanged? At this point we cannot answer the question definitively. However, we can try to verify that combining the above transformations will not yield new transformations. Specifically, if we let $\sigma * \tau$ denote the application of τ followed by σ we obtain the *symmetry composition table* in Table 10.2[1]. Thus the set of transformations $\{\varepsilon, \alpha, \beta, \gamma\}$ is *closed* under the composition operator $*$.

Table 10.1. Truth table for example Boolean function (10.1)

a	b	c	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

Table 10.2. Symmetry composition table for example function (10.1). The entry in row r and column c represents the result of applying symmetry c followed by symmetry r.

$r * c$	ε	α	β	γ
ε	ε	α	β	γ
α	α	ε	γ	β
β	β	γ	ε	α
γ	γ	β	α	ε

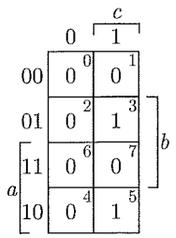
To visualize this function's symmetries, consider its Karnaugh map (K-map) in Table 10.3. The map clearly shows that the function's solution space (the 8 possible truth assignments to the function's three propositional variables) consists of two symmetric subspaces. Specifically, the distribution of the function's 1s and 0s in the subspace where $a = 0$ is identical to their distribution in the subspace where $a = 1$. Stated another way, the function's 8 possible truth assignments can be collapsed to just these 4 equivalence classes: $\{0, 6\}$, $\{1, 7\}$, $\{2, 4\}$, and $\{3, 5\}$. The assignments in each such class are equivalent in the sense that they yield the same function value (either 0 or 1). Thus, to determine the satisfiability of the function, it is sufficient to consider only one such assignment from each equivalence class. This can be accomplished by conjoining the function with a constraint that filters out all but the chosen representative assignment from each class. A convenient filter for this example is either the constraint (a), which restricts the search for a satisfying assignment to the bottom 4 cells of the map, or the constraint (a'), which restricts the search to the top 4 cells of the map[2]. These filters are referred to as *symmetry-breaking predicates* (SBPs for short) because they serve to block a search algorithm from exploring symmetric assignments in the solution space.

The rest of this chapter is divided into eight sections. In Sec. 10.2 we cover some preliminaries related to Boolean algebra and set partitions. Sec. 10.3 introduces the basics of group theory necessary for representing and manipulating the symmetries of Boolean functions. This is further articulated in Sec. 10.4 for the CNF representation of such functions. In Sec. 10.5 we briefly describe the colored graph automorphism problem and the basic algorithm for solving it. This sets the stage, in Sec. 10.6, for the discussion on detecting the symmetries of CNF formulas by reduction to graph automorphism. The construction of symmetry breaking predicates is covered in Sec. 10.7, and Sec. 10.8 summarizes the whole symmetry detection and breaking flow, as well as offering some ideas on an alternative approach that requires further exploration. The chapter concludes in Sec. 10.9 with a historical review of relevant literature.

[1] Some authors define composition using the opposite order: $\sigma * \tau$ means "apply σ followed by τ." This does not matter as long as the operation order is defined consistently.

[2] Other, less convenient, filters include $(a'c' + ac)$ which selects assignments $\{0, 2, 5, 7\}$ and excludes assignments $\{1, 3, 4, 6\}$.

Table 10.3. K-map for example function (10.1). The small numbers in the map cells indicate the decimal index of the corresponding minterm (variable assignment) assuming that a is the most significant bit and c is the least significant bit.



10.2. Preliminaries

Since we will be mostly concerned with symmetries in the context of Boolean satisfiability, it is useful to establish in this section the vocabulary of Boolean algebra to which we will make frequent reference. Specifically, we will be exclusively dealing with the 2-valued Boolean algebra, also known as propositional logic, which we define as follows:

Definition 10.2.1. (2-Valued Boolean Algebra) The 2-valued Boolean algebra is the algebraic structure $\langle \{0,1\}, \cdot, +, ' \rangle$ where $\{0,1\}$ is the set of *truth values*, and where the binary operators $\cdot$ and $+$, and the unary operator $'$ are defined by the following rules:

- AND (logical conjunction): $0 \cdot 0 = 0 \cdot 1 = 1 \cdot 0 = 0, 1 \cdot 1 = 1$
- OR (logical disjunction): $0 + 0 = 0, 0 + 1 = 1 + 0 = 1 + 1 = 1$
- NOT (logical negation): $0' = 1, 1' = 0$

We will, on occasion, use other symbols to indicate these operators such as $\wedge$ for AND, $\vee$ for OR, and $\neg$ for NOT. In addition, we will follow custom by indicating the AND operator with a simple juxtaposition of its arguments.

A Boolean *variable* is a variable whose domain is the set $\{0,1\}$. Although not formally synonymous, we may at times refer to such variables as *binary* or as *propositional*. A Boolean *literal* is either a Boolean variable or the negation of a Boolean variable. The domain of a vector of n Boolean variables is the set $\{0,1\}^n$ whose elements are the 2^n ordered n-tuples, or combinations, of 0 and 1. In other words,

$$\{0,1\}^n = \left\{ \underbrace{(0,0,\ldots,0,0)}_{n \text{ bits}}, \underbrace{(0,0,\ldots,0,1)}_{n \text{ bits}}, \ldots, \underbrace{(1,1,\ldots,1,1)}_{n \text{ bits}} \right\}$$

Each of these combinations can also be interpreted naturally as an unsigned n-bit binary integer. Expressing each of these integers in decimal notation, we obtain the following equivalent representation of this domain:

$$\{0,1\}^n = \{0,1,2,\ldots,2^n-1\}$$

We will refer to elements of this set as *truth assignments* or "points" in the n-dimensional Boolean space and identify each by its decimal "index".

Individual Boolean variables will be denoted by lower-case letters, possibly with a numeric subscript, e.g., a, b, x_3, etc. When speaking about the literals of a particular variable we will place a dot over the variable symbol. Thus, the literal $\dot{a}$ stands for either a or a'. Vectors of Boolean variables will be denoted by upper case letters and, as above, interpreted as unsigned binary integers. Thus, $X = (x_{n-1}, x_{n-2}, \ldots, x_0)$ is a vector of n Boolean variables representing an unsigned n-bit integer whose most significant bit is x_{n-1} and whose least significant bit is x_0. An assignment $X = i$ to an n-variable Boolean vector where $0 \leq i \leq 2^n - 1$ is short-hand for assigning to the elements of X the appropriate bits of the binary integer corresponding to i. Thus, for $n = 3$, $X = 5$ means $x_2 = 1$, $x_1 = 0$ and $x_0 = 1$.

An n-variable Boolean function is defined by the map $\{0,1\}^n \rightarrow \{0,1\}$, i.e., each of the 2^n points in its domain is mapped to one of the two binary values. There are two classes of elementary Boolean functions: minterms and maxterms.

Definition 10.2.2. (Minterms and Maxterms) The *ith n-variable minterm function*, denoted by $m_i(X)$, maps point i in the n-dimensional Boolean space to 1, and maps all other points to 0. The *ith n-variable maxterm function*, denoted by $M_i(X)$, maps point i in the n-dimensional Boolean space to 0, and maps all other points to 1. Algebraically, an n-variable minterm function is an AND expression of n distinct literals, whereas an n-variable maxterm function is an OR expression of n distinct literals.

AND and OR expressions (not necessarily consisting of all n literals) are commonly referred to as *product* and *sum terms*, respectively.

Minterm and maxterm functions serve as building blocks for representing any n-variable Boolean function. For example, the function specified in Table 10.1 has the following *sum-of-minterm* and *product-of-maxterm* representations:

$$\begin{aligned} f(a,b,c) &= m_3 + m_5 = a'bc + ab'c \\ &= M_0 \cdot M_1 \cdot M_2 \cdot M_4 \cdot M_6 \cdot M_7 \\ &= (a+b+c)(a+b+c')(a+b'+c)(a'+b+c)(a'+b'+c)(a'+b'+c') \end{aligned}$$

Examination of the sum-of-minterms expression shows that it evaluates to 1 for combinations 3 and 5 on variables (a,b,c) and evaluates to 0 for all other combinations. Informally, we say that function f consists of minterms 3 and 5. Alternatively, we can say that f consists of maxterms 0, 1, 2, 4, 6, and 7. A commonly-used short-hand for representing Boolean functions this way is the following:

$$f(a,b,c) = \sum\nolimits_{(a,b,c)} (3,5) = \prod\nolimits_{(a,b,c)} (0,1,2,4,6,7)$$

Definition 10.2.3. (Implication) Given two Boolean functions $f(X)$ and $g(X)$, we say that $g(X)$ *implies* $f(X)$, written $g(X) \rightarrow f(X)$, if all of g's minterms are also minterms of f.

Definition 10.2.4. (Implicants and Implicates) A product term $p(X)$ is an *implicant* of a function $f(X)$ if $p(X) \rightarrow f(X)$. A sum term $s(X)$ is an *implicate* of a function $f(X)$ if $f(X) \rightarrow s(X)$.

A function can have many implicants and implicates. In particular a function's implicants include its minterms and its implicates include its maxterms.

Definition 10.2.5. (Disjunctive and Conjunctive Normal Forms) A *disjunctive normal form* (DNF) expression of a Boolean function is an OR of implicants. It is also known as a sum-of-products (SOP). A *conjunctive normal form* (CNF) expression of a Boolean function is an AND of implicates. It is also known as a product-of-sums (POS). It is customary to refer to the implicates in a CNF representation as *clauses*.

Definition 10.2.6. (Minimal DNF and CNF Expressions) The *cost* of a DNF or CNF expression $\varphi(X)$ is the tuple $(\text{terms}(\varphi), \text{literals}(\varphi))$ where $\text{terms}(\varphi)$ and $\text{literals}(\varphi)$ are, respectively, the number of terms (implicants or implicates) and total number of literals in the expression. We say that expression φ is *cheaper* than expression ϑ if $\text{terms}(\varphi) \leq \text{terms}(\vartheta)$ and $\text{literals}(\varphi) < \text{literals}(\vartheta)$. A DNF expression $\varphi(X)$ is minimal for function $f(X)$ if there are no other DNF expressions for $f(X)$ that are cheaper than $\varphi(X)$. Similarly, A CNF expression $\varphi(X)$ is minimal for function $f(X)$ if there are no other CNF expressions for $f(X)$ that are cheaper than $\varphi(X)$. Note that this definition of cost induces a partial order on the set of equivalent DNF (resp. CNF) expressions of a Boolean function.

Definition 10.2.7. (Prime Implicants and Prime Implicates) A *prime implicant* $p(X)$ of a function $f(X)$ is an implicant that is not contained in any other implicant of $f(X)$. In other words, $p(X)$ is a prime implicant of $f(X)$ if $p(X) \rightarrow f(X)$ and $p(X) \nrightarrow q(X)$ where $q(X)$ is an implicant of $f(X)$. A *prime implicate* $s(X)$ of a function $f(X)$ is an implicate that does not contain any other implicate of $f(X)$. In other words, $s(X)$ is a prime implicate of $f(X)$ if $f(X) \rightarrow s(X)$ and $t(X) \nrightarrow s(X)$ where $t(X)$ is an implicate of $f(X)$.

Theorem 10.2.1. (Prime Implicant/Prime Implicate Theorem [Qui52, Qui55, Qui59]) Any minimal DNF expression of a function $f(X)$ can only consist of prime implicants. Similarly, any minimal CNF expression can only consist of prime implicates. Minimal expressions are not necessarily unique.

Definition 10.2.8. (leq predicate) The *less-than-or-equal* predicate in the n-dimensional Boolean space is defined by the map

$$\begin{aligned} \text{leq}\,(X,Y) &= X \leqslant Y \\ &= \bigwedge_{i\in[0,n-1]} \left(\left(\bigwedge_{j\in[i+1,n-1]} (x_j = y_j) \right) \rightarrow (x_i \leqslant y_i) \right) \end{aligned} \tag{10.5}$$

This is a common arithmetic function that is usually implemented in a multi-level circuit similar to that shown in Fig. 10.2. In comparing the circuit to equation (10.5), note that $(x \rightarrow y) = (x \leq y) = (x' + y)$ and $(x = y)' = (x \oplus y)$, where $\oplus$ denotes *exclusive OR*. Note also that the leq predicate imposes a total ordering on the 2^n truth assignments that conforms to their interpretation as unsigned integers. As will become clear later, the leq predicate provides one mechanism for breaking symmetries.

We close with a brief mention of set partitions since they arise in the context of describing and detecting the symmetries of a CNF formula.

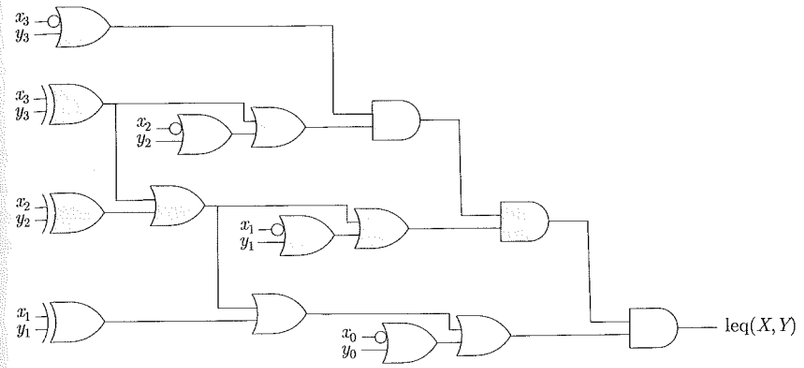


Figure 10.2. A 4-bit comparator circuit

Definition 10.2.9. (Partition) A partition π of a set X is a set of non-empty subsets of X, $\{X_i | X_i \subseteq X, i = 1, \ldots, k\}$, such that:

$$X_i \cap X_j = \emptyset, \text{ for } i \neq j$$
$$\bigcup_{1\leqslant i\leqslant k} X_i = X$$

The elements of π are variously referred to as the *blocks*, *classes* or *cells* of the partition. A partition is *discrete* if all of its cells are singleton sets, and *unit* if it has only one cell (the whole set).

Two elements $x, y \in X$ are equivalent under partition π, written as $x \sim y$, if $x, y \in X_i$ for some $i \in [1, k]$.

Definition 10.2.10. (Lattice of Partitions [Can01]) Given two partitions π and π' of a set X, we say that π' is *finer* than π, written as $\pi' \leq \pi$, if every cell of π' is contained in some cell of π. The relation "is finer than" defines a *lattice* on the set of partitions whose top elements is the unit partition and whose bottom element is the discrete partition. Given two partitions π_1 and π_2, their *intersection*, $\pi_1 \cap \pi_2$, is the *largest* partition that refines both. Dually, the *union* of π_1 and π_2, $\pi_1 \cup \pi_2$, is the *smallest* partition which is refined by both.

Example 10.2.1. Given $X = \{1, 2, \ldots, 10\}$ and the two partitions

$$\pi_1 = \{\{1,2,4\}, \{3\}, \{5\}, \{6,7,8\}, \{9,10\}\}$$
$$\pi_2 = \{\{1,3,4\}, \{2\}, \{5,6,7,8\}, \{9\}, \{10\}\}$$

their intersection and union are

$$\pi_1 \cap \pi_2 = \{\{1,4\}, \{2\}, \{3\}, \{5\}, \{6,7,8\}, \{9\}, \{10\}\}$$
$$\pi_1 \cup \pi_2 = \{\{1,2,3,4\}, \{5,6,7,8\}, \{9,10\}\}$$

10.3. Group Theory Basics

The study of symmetries in Boolean functions and their various representations is greatly facilitated by understanding some basic notions from the vast field of group theory. In this section we cover only those aspects of group theory that are necessary to follow the development, in subsequent sections, of the symmetry detection and breaking approaches for CNF representations of Boolean functions. More comprehensive treatments of group theory are available in standard textbooks; a particularly accessible source is [Fra00] which we liberally borrow from in what follows.

10.3.1. Groups

We begin with the definition of a group as an abstract algebraic structure with certain properties.

Definition 10.3.1. (Group) A group is a structure $\langle G, * \rangle$, where G is a (non-empty) set that is closed under a binary operation $*$, such that the following axioms are satisfied:

- The operation is *associative*: for all $x, y, z \in G, (x * y) * z = x * (y * z)$
- There is an *identity* element $e \in G$ such that: for all $x \in G, e * x = x * e = x$
- Every element $x \in G$ has an inverse $x^{-1} \in G$ such that: $x * x^{-1} = x^{-1} * x = e$

We will typically refer to the group G, rather than to the structure $\langle G, * \rangle$, with the understanding that there is an associated binary operation on the set G. Furthermore, it is customary to write xy instead of $x * y$ and x^2 instead of $x * x$ much like we do with the multiplication operation on numbers.

This definition leads to a number of interesting properties that are easy to prove, including:

- **Uniqueness of the identity**: there is only one element $e \in G$ such that $e * x = x * e = x$ for all $x \in G$, and
- **Uniqueness of the inverse**: for each $x \in G$, there is only one element $x^{-1} \in G$ such that $x * x^{-1} = x^{-1} * x = e$.

The definition also does not restrict a group to be finite. However, for our purposes it is sufficient to focus on finite groups.

Definition 10.3.2. (Group Order) If G is a finite group, its *order* $|G|$ is the number of elements in (i.e., the cardinality of) the set G.

Under these two definitions, the set of transformations introduced in Sec. 10.1 is easily seen to be a group of order 4 with group operation $*$ as defined in Table 10.2. The identity element is the "do nothing" transformation ε and each of the four elements in the group is its own inverse. The associativity of the operation can be verified by enumerating all possible triples from $\{\varepsilon, \alpha, \beta, \gamma\}^3$.[3]

[3] Such enumeration is impractical or even infeasible for large groups. In general, proving that the group operation is associative is typically done by showing that the group is isomorphic (see Def. 10.3.3) to another group whose operation is *known* to be associative.

This group is known as the Klein 4-group V and is one of only two groups of order 4 [Fra00, p. 67].

Definition 10.3.3. (Group Isomorphism) Let $\langle G, * \rangle$ and $\langle G', *' \rangle$ be two groups. An *isomorphism* of G with G' is a one-to-one function ϕ mapping G onto G' such that:

$$\phi(x * y) = \phi(x) *' \phi(y) \text{ for all } x, y \in G$$

If such a map exists, then G and G' are *isomorphic groups* which we denote by $G \simeq G'$.

To illustrate group isomorphism, consider the following set of all possible negations (complementations) of two binary literals $\dot{a}, \dot{b}$:

Label	Negation
$\dot{a}\dot{b}$	Don't negate either literal
$\dot{a}'\dot{b}$	Only negate $\dot{a}$
$\dot{a}\dot{b}'$	Only negate $\dot{b}$
$\dot{a}'\dot{b}'$	Negate both $\dot{a}$ and $\dot{b}$

The labels in the above table should be viewed as mnemonics that indicate the particular transformation on the two literals: a prime on a literal means the literal should be complemented by the transformation, and the absence of a prime means the literal should be left unchanged. As such, these transformations can be combined. For example, negating both literals followed by negating only $\dot{a}$ yields the same result as negating just $\dot{b}$. Denoting this operation by $*'$ we obtain the composition table in Table 10.4. Clearly, as defined, the set $\{\dot{a}\dot{b}, \dot{a}'\dot{b}, \dot{a}\dot{b}', \dot{a}'\dot{b}'\}$ along with the operation $*'$ is a group. Let's denote this group by $\mathcal{N}_2$ (for the *group of negations on two binary literals*). Close examination of Table 10.2 and Table 10.4 shows that, other than the particular labels used, they have identical structures. Specifically, we have the following map between the Klein 4-group V and the group of negations $\mathcal{N}_2$:

$$\begin{array}{ccc} x \in V & \leftrightarrow & \phi(x) \in \mathcal{N}_2 \\ \varepsilon & \leftrightarrow & \dot{a}\dot{b} \\ \alpha & \leftrightarrow & \dot{a}'\dot{b} \\ \beta & \leftrightarrow & \dot{a}\dot{b}' \\ \gamma & \leftrightarrow & \dot{a}'\dot{b}' \end{array}$$

Thus V and $\mathcal{N}_2$ are isomorphic groups. In what follows we will use $\mathcal{N}_n$ to denote the group of negations on n literals. This group has order 2^n.

10.3.2. Subgroups

Subgroups provide a way to understand the structure of groups.

Definition 10.3.4. (Subgroup) A non-empty subset H of a group G that is closed under the binary operation of G is itself a group and is referred to as a *subgroup* of G. We indicate that H is a subgroup of G by writing $H \leq G$. Additionally, $H < G$ shall mean that $H \leq G$ but $H \neq G$.

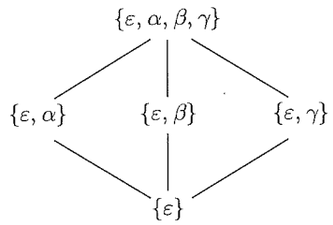


Figure 10.3. Lattice diagram for the Klein 4-group V showing its subgroups.

Definition 10.3.5. (Proper and Trivial Subgroups) If G is a group, then the subgroup consisting of G itself is the *improper subgroup* of G. All other subgroups are *proper subgroups*. The group $\{e\}$ is the *trivial subgroup* of G. All other subgroups are *nontrivial*.

We illustrate the notion of subgroups for the Klein 4-group introduced earlier. Specifically, this group has four non-trivial subgroups: $\{\varepsilon, \alpha\}, \{\varepsilon, \beta\}, \{\varepsilon, \gamma\}$, as well as the whole group itself. These groups are related according to the lattice structure shown in Fig. 10.3.

A subgroup H of a group G induces a partition of G whose cells are referred to as the *cosets* of H:

Definition 10.3.6. (Cosets) Let H be a subgroup of a group G. The *left coset* of H containing $x \in G$ is the subset $xH = \{xy | y \in H\}$ and the *right coset* of H containing $x \in G$ is the subset $Hx = \{yx | y \in H\}$.

For example, the left cosets of $H = \{\varepsilon, \alpha\}$ in the Klein-4 group $\{\varepsilon, \alpha, \beta, \gamma\}$ yield the partition $\{\{\varepsilon, \alpha\}, \{\beta, \gamma\}\}$.

It is easily shown that the number of elements in each coset of a subgroup H of a group G is the same as the number of elements of H [Fra00, p. 124]. This immediately leads to the following fundamental theorem of group theory:

Theorem 10.3.1. (Theorem of Lagrange) Let H be a subgroup of a finite group G. Then $|H|$ is a divisor of $|G|$.

10.3.3. Group Generators

Theorem 10.3.2. (Cyclic Subgroup) Let G be a group and let $x \in G$. Then

$$H = \{x^n | n \in \mathbb{Z}\}$$

Table 10.4. Composition table for $\mathcal{N}_2$

$*'$	$\dot{a}\dot{b}$	$\dot{a}'\dot{b}$	$\dot{a}\dot{b}'$	$\dot{a}'\dot{b}'$
$\dot{a}\dot{b}$	$\dot{a}\dot{b}$	$\dot{a}'\dot{b}$	$\dot{a}\dot{b}'$	$\dot{a}'\dot{b}'$
$\dot{a}'\dot{b}$	$\dot{a}'\dot{b}$	$\dot{a}\dot{b}$	$\dot{a}'\dot{b}'$	$\dot{a}\dot{b}'$
$\dot{a}\dot{b}'$	$\dot{a}\dot{b}'$	$\dot{a}'\dot{b}'$	$\dot{a}\dot{b}$	$\dot{a}'\dot{b}$
$\dot{a}'\dot{b}'$	$\dot{a}'\dot{b}'$	$\dot{a}\dot{b}'$	$\dot{a}'\dot{b}$	$\dot{a}\dot{b}$

i.e., the set of all (not necessarily distinct) powers of x, is a subgroup of G and is the smallest subgroup of G that contains x. This is referred to as the *cyclic subgroup of G generated by x*, and is denoted by $\langle x \rangle$.

Definition 10.3.7. (Generator; Cyclic Group) An element x of a group G *generates* G and is a *generator* for G if $\langle x \rangle = G$. A group G is *cyclic* if there is some element $x \in G$ that generates G.

Definition 10.3.8. (Group Generators) Let G be a group and let $H \subset G$ be a subset of the group elements. The smallest subgroup of G containing H is the *subgroup generated by H*. If this subgroup is all of G, then H *generates* G and the elements of H are *generators* of G. A generator is *redundant* if it can be expressed in terms of other generators. A set of generators for a group G is *irredundant* if it does not contain redundant generators.

An important property of irredundant generator sets of a finite group is that they provide an extremely compact representation of the group. This follows directly from the Theorem of Lagrange and the definition of irredundant generator sets:

Theorem 10.3.3. Any irredundant set of generators for a finite group G, such that $|G| > 1$, contains at most $\log_2 |G|$ elements.

Proof. Let $\{x_i \in G | 1 \leq i \leq n\}$ be a set of n irredundant generators for group G and consider the sequence of subgroups $G_1, G_2, \ldots, G_n$ such that group G_i is generated by $\{x_j \in G | 1 \leq j \leq i\}$. Clearly, $G_1 < G_2 < \cdots < G_n$ because of our assumption that the generators are irredundant. By the theorem of Lagrange, $|G_n| \geq 2|G_{n-1}| \geq \cdots \geq 2|G_2| \geq 2|G_1|$, i.e., $|G_n| \geq 2^{n-1}|G_1|$. Noting that $G_n = G$ and that $|G_1| \geq 2$, we get $|G| \geq 2^n$. Thus, $n \leq \log_2 |G|$. □

Returning to the Klein 4-group, we note that it has three sets of irredundant generators: $\{\alpha, \beta\}$, $\{\alpha, \gamma\}$, and $\{\beta, \gamma\}$. It can also be generated by the set $\{\alpha, \beta, \gamma\}$, but this set is redundant since any of its elements can be obtained as the product of the other two.

10.3.4. Permutation Groups

We turn next to an important class of groups, namely groups of permutations.

Definition 10.3.9. (Permutation) A permutation of a set A is a function $\phi : A \to A$ that is both one to one and onto (a bijection).

Permutations can be "multiplied" using function composition:

$$\sigma\tau\,(a) \equiv (\sigma \circ \tau)\,(a) = \sigma\,(\tau\,(a))$$

where σ and τ are two permutations of A and $a \in A$. The resulting "product" is also a permutation of A, since it is easily shown to be one to one and onto, and leads us to the following result:

Theorem 10.3.4. (Permutation Group) Given a non-empty set A, let S_A be the set of all permutations of A. Then S_A is a group under permutation multiplication.

$$\sigma = \begin{pmatrix} 1\,2\,3\,4\,5\,6\,7\,8 \\ 8\,2\,6\,4\,5\,3\,7\,1 \end{pmatrix} \quad \tau = \begin{pmatrix} 1\,2\,3\,4\,5\,6\,7\,8 \\ 1\,7\,5\,4\,6\,3\,8\,2 \end{pmatrix}$$
$$\sigma\tau = \begin{pmatrix} 1\,2\,3\,4\,5\,6\,7\,8 \\ 8\,7\,5\,4\,3\,6\,1\,2 \end{pmatrix} \quad \tau\sigma = \begin{pmatrix} 1\,2\,3\,4\,5\,6\,7\,8 \\ 2\,7\,3\,4\,6\,5\,8\,1 \end{pmatrix}$$

Figure 10.4. Example permutations of the set $A = \{1, 2, 3, 4, 5, 6, 7, 8\}$ along with their products.

Definition 10.3.10. (Symmetric Group) The group of all permutations of the set $\{1, 2, \ldots, n\}$ $(n \geq 1)$ is the *symmetric group on n letters*, and is denoted by S_n.

Fig. 10.4 shows two example permutations of an 8-element set along with their products. Note that, in general, permutation multiplication is not commutative, i.e., $\sigma\tau \neq \tau\sigma$. The permutations in this figure are presented in a *tabular format* that explicitly shows how each element of the underlying set is mapped to a distinct element of the same set. For instance, element 3 is mapped by σ to 6: $\sigma(3) = 6$. This explicit format becomes cumbersome, however, when the cardinality of the set A is large, and especially when a permutation maps many of the elements of A to themselves. In such cases, the more compact *cycle notation* is preferred. Using cycle notation, the permutations in Fig. 10.4 can be expressed succinctly as follows:

$$\sigma = (1, 8)\,(3, 6) \qquad \tau = (2, 7, 8)\,(3, 5, 6)$$
$$\sigma\tau = (1, 8, 2, 7)\,(3, 5) \quad \tau\sigma = (1, 2, 7, 8)\,(5, 6)$$

Each such permutation is a product of *disjoint cycles*, where a cycle $(a, b, c, \ldots, z)$ is understood to mean that the permutation maps a to b, b to c, and so on, finally mapping the last element in the cycle z back to a. An element that does not appear in a cycle is understood to be left fixed (mapped to itself) by the cycle. The *length* of a cycle is the number of elements in the cycle; a cycle with k elements will be referred to as a k-cycle.

Definition 10.3.11. (Permutation Support) Given a permutation σ of a set A, its support consists of those elements of A that are not mapped to themselves by σ:

$$\text{supp}(\sigma) = \{a \in A | \sigma(a) \neq a\} \tag{10.6}$$

In other words, $\text{supp}(\sigma)$ are those elements of A that appear in σ's cycle representation.

Permutation groups are in, some sense, fundamental because of the following result:

Theorem 10.3.5. (Cayley's Theorem) Every group is isomorphic to a group of permutations.

To illustrate, consider the following permutations defined on the Klein 4-group

elements:

$$\lambda_\varepsilon = \begin{pmatrix} \varepsilon\ \alpha\ \beta\ \gamma \\ \varepsilon\ \alpha\ \beta\ \gamma \end{pmatrix} \quad \lambda_\alpha = \begin{pmatrix} \varepsilon\ \alpha\ \beta\ \gamma \\ \alpha\ \varepsilon\ \gamma\ \beta \end{pmatrix}$$
$$\lambda_\beta = \begin{pmatrix} \varepsilon\ \alpha\ \beta\ \gamma \\ \beta\ \gamma\ \varepsilon\ \alpha \end{pmatrix} \quad \lambda_\gamma = \begin{pmatrix} \varepsilon\ \alpha\ \beta\ \gamma \\ \gamma\ \beta\ \alpha\ \varepsilon \end{pmatrix}$$

The set of these four permutations along with permutation multiplication as the binary operation on the set satisfy the definition of a group. Additionally, the Klein 4-group is isomorphic to this permutation group using the mapping $\lambda_\varepsilon \leftrightarrow \varepsilon, \lambda_\alpha \leftrightarrow \alpha, \lambda_\beta \leftrightarrow \beta, \lambda_\gamma \leftrightarrow \gamma$.

It is useful at this point to recast the group of negations $\mathcal{N}_n$ introduced earlier as a group of permutations of the set of positive and negative literals on n Boolean variables. Specifically, $\mathcal{N}_2$ can be expressed as a group of permutations of the set $\{a, a', b, b'\}$. Denoting these permutations by $\eta_\emptyset, \eta_{\{a,a'\}}, \eta_{\{b,b'\}}$ and $\eta_{\{a,a',b,b'\}}$ where η_S indicates that the literals in the set S should be complemented, we can readily write

$$\eta_\emptyset = \begin{pmatrix} a\ a'\ b\ b' \\ a\ a'\ b\ b' \end{pmatrix} \quad \eta_{\{a,a'\}} = \begin{pmatrix} a\ a'\ b\ b' \\ a'\ a\ b\ b' \end{pmatrix}$$
$$\eta_{\{b,b'\}} = \begin{pmatrix} a\ a'\ b\ b' \\ a\ a'\ b'\ b \end{pmatrix} \quad \eta_{\{a,a',b,b'\}} = \begin{pmatrix} a\ a'\ b\ b' \\ a'\ a\ b'\ b \end{pmatrix}$$

We should note that since the literals of each variable are complements of each other, the above permutations can be re-expressed by listing only the positive literals in the top row of each permutation (and by dropping the negative literals from the subscripts of η):

$$\eta_\emptyset = \begin{pmatrix} a\ b \\ a\ b \end{pmatrix}, \eta_{\{a\}} = \begin{pmatrix} a\ b \\ a'\ b \end{pmatrix}, \eta_{\{b\}} = \begin{pmatrix} a\ b \\ a\ b' \end{pmatrix}, \eta_{\{a,b\}} = \begin{pmatrix} a\ b \\ a'\ b' \end{pmatrix} \tag{10.7}$$

These same permutations can be expressed most succinctly in cycle notation as follows:

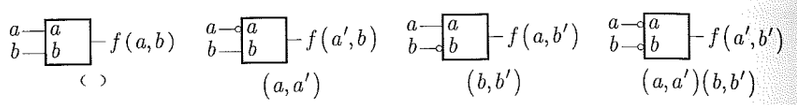
$$\eta_\emptyset = (), \eta_{\{a\}} = (a, a'), \eta_{\{b\}} = (b, b'), \eta_{\{a,b\}} = (a, a')\,(b, b') \tag{10.8}$$

where the identity permutation is denoted by $()$. In both (10.7) and (10.8) it is implicitly understood that the underlying set whose elements are being permuted is the set of literals $\{a, a', b, b'\}$.

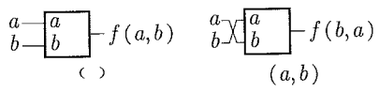
10.3.5. Group of Negations and Permutations

We are now ready to discuss the group of most relevance to our study of symmetry and satisfiability, namely the group of negations *and* permutations of the literals of an n-variable CNF formula. We will denote this group by $\mathcal{NP}_n$ and refer to it as the group of negations and permutations of n variables, with the understanding that its underlying set consists of $2n$ elements that correspond to the literals of n Boolean variables. Any permutation $\pi \in \mathcal{NP}_n$ must satisfy *Boolean consistency* which requires that when literal $\dot{x}$ is mapped to literal $\dot{y}$, its negated literal $\dot{x}'$ is also simultaneously mapped to literal $\dot{y}'$, i.e.,

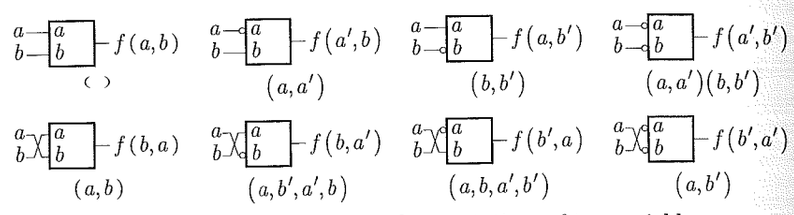
$$\pi(\dot{x}_i') = [\pi(\dot{x}_i)]' \tag{10.9}$$



(a) Group N_2: negations of two variables



(b) Group P_2: permutations of two variables



(c) Group NP_2: negations *and* permutations of two variables

Figure 10.5. Groups of (a) negations, (b) permutations, and (c) mixed negations *and* permutations of two variables.

It is important to note that a group $G \leq S_{2n}$ that permutes the set of $2n$ literals without any restrictions may contain permutations that violate Boolean consistency; e.g. $(a, b')(a', c)$.

Group $\mathcal{NP}_n$ has two important subgroups: the group of negations of n variables $\mathcal{N}_n$, and the group of permutations of n variables $\mathcal{P}_n$[4]. Fig. 10.5 gives pictorial representations of $\mathcal{NP}_2$ as well as of its constituent subgroups $\mathcal{N}_2$ and $\mathcal{P}_2$. By convention, when expressing the elements of these groups we will omit the mappings implied by Boolean consistency: thus we will write (a, b') rather than $(a, b')(a', b)$, the latter being implied by the former.

From Fig. 10.5 it is clear that each element of $\mathcal{NP}_2$ can be expressed as a pair (η, π) where $\eta \in \mathcal{N}_2$ and $\pi \in \mathcal{P}_2$. Furthermore, since both $\mathcal{N}_2$ and $\mathcal{P}_2$ are permutation groups, the element $(\eta, \pi) \in \mathcal{NP}_2$ is naturally interpreted as the composition of a negation η followed by a permutation π. For example, $((b, b'), (a, b))$ corresponds to negating b followed by swapping a and b yielding the permutation $(a, b) * (b, b') = (a, b, a', b')$ where the permutation composition operator $*$ is shown explicitly for added clarity.

More generally, the group $\mathcal{NP}_n$ is the *semi-direct product* [Har63] of $\mathcal{N}_n$ and $\mathcal{P}_n$, typically denoted by $\mathcal{N}_n \rtimes \mathcal{P}_n$, whose underlying set is the Cartesian product

[4]We prefer to denote the group of permutations of n variables by $\mathcal{P}_n$ rather than by S_n to emphasize that its underlying set is the set of $2n$ literals and that its elements obey Boolean consistency (10.9). Other than that, $\mathcal{P}_n$ and S_n are isomorphic groups.

$\mathcal{N}_n \times \mathcal{P}_n = \{(\eta, \pi) | \eta \in \mathcal{N}_n, \pi \in \mathcal{P}_n\}$ and whose group operation $*$ is defined by:

$$(\eta_1, \pi_1) * (\eta_2, \pi_2) = \pi_1 \eta_1 \pi_2 \eta_2 \tag{10.10}$$

For example, $((a, a'), ()) * ((a, a'), (a, b)) = ()(a, a')(a, b)(a, a') = (a, b')$. Clearly, the order of $\mathcal{NP}_n$ is $2^n \cdot n!$.

Returning to the example function in Sec. 10.1, we can express the four transformations that leave it invariant as the semi-direct product group

$$\begin{aligned} &\{(), (a, a')(b, b')\} \rtimes \{(), (a, b)\} \\ &= \{((), ()), ((a, a')(b, b'), ()), ((), (a, b)), ((a, a')(b, b'), (a, b))\} \\ &= \{(), (a, a')(b, b'), (a, b), (a, b')\} \end{aligned} \tag{10.11}$$

Note that these four permutations do in fact constitute a group which, again, is easily seen to be isomorphic to the Klein 4-group.

10.3.6. Group Action on a Set

Finally, let's introduce the notion of a group *action* which is fundamental to the development of symmetry-breaking predicates for CNF formulas of Boolean functions.

Definition 10.3.12. (Group Action) An *action* of a group G on a set S is a map $G \times S \to S$ such that:

- $es = s$ for all $s \in S$
- $(g_1 g_2)(s) = g_1(g_2 s)$ for all $s \in S$ and all $g_1, g_2 \in G$.

The group action that we will be primarily concerned with is the action of a group of permutations of the literals of a Boolean function on the set of points (i.e., truth assignments) in the function's domain. For example, Table 10.5 shows the action of the group of permutations in (10.11) on the set of 8 truth assignments to the variables a, b, and c. It is customary to use superscript notation to indicate the action of a group element on an element of the set being acted on. Thus, e.g., $1^{\pi_1} = 7$. The notation extends naturally to collections (sets, vectors, graphs, etc.), e.g., $\{0, 4, 6, 7\}^{\pi_3} = \{0^{\pi_3}, 4^{\pi_3}, 6^{\pi_3}, 7^{\pi_3}\} = \{6, 4, 0, 1\}$.

A group action on a set induces an equivalence partition on the set according to the following theorem:

Theorem 10.3.6. (Group-Induced Equivalence Partition; Orbits) The action of a group G on a set S induces an equivalence relation on the set. Specifically, for $s_1, s_2 \in S$, let $s_1 \sim s_2$ if and only if there exists $g \in G$ such that $g s_1 = g_2$. Then $\sim$ is an equivalence relation on S that partitions it into equivalence classes referred to as the *orbits* of S under G. The orbits of S under a particular group element $g \in G$ are those that satisfy $s_1 \sim s_2$ if and only if $s_2 = g^n s_1$ for some $n \in \mathbb{Z}$.

Table 10.5. Action table of the group $\{(), (a,a')(b,b'), (a,b), (a,b')\}$ on the set $\{0,1,2,3,4,5,6,7\}$ of truth assignments to a, b, c.

	0	1	2	3	4	5	6	7
$\pi_0 = ()$	0	1	2	3	4	5	6	7
$\pi_1 = (a,a')(b,b')$	6	7	4	5	2	3	0	1
$\pi_2 = (a,b)$	0	1	4	5	2	3	6	7
$\pi_3 = (a,b')$	6	7	2	3	4	5	0	1

Using our running example, we can "extract" the orbits of the four permutations in Table 10.5 by inspection:

$$\begin{aligned}
\text{orbits}\,(S,\pi_0) &= \{0\mapsto 0, 1\mapsto 1, 2\mapsto 2, 3\mapsto 3, 4\mapsto 4, 5\mapsto 5, 6\mapsto 6, 7\mapsto 7\}\\
\text{orbits}\,(S,\pi_1) &= \{0\mapsto 6\mapsto 0, 1\mapsto 7\mapsto 1, 2\mapsto 4\mapsto 2, 3\mapsto 5\mapsto 3\}\\
\text{orbits}\,(S,\pi_2) &= \{0\mapsto 0, 1\mapsto 1, 2\mapsto 4\mapsto 2, 3\mapsto 5\mapsto 3, 6\mapsto 6, 7\mapsto 7\}\\
\text{orbits}\,(S,\pi_3) &= \{0\mapsto 6\mapsto 0, 1\mapsto 7\mapsto 1, 2\mapsto 2, 3\mapsto 3, 4\mapsto 4, 5\mapsto 5\}
\end{aligned} \tag{10.12}$$

where $S = \{0,1,2,3,4,5,6,7\}$ and $x \mapsto y$ means that x is mapped to y by the corresponding permutation. Using the more familiar partition notation, these same orbits can be equivalently expressed as follows:

$$\begin{aligned}
\text{partition}\,(S,\pi_0) &= \{\{0\},\{1\},\{2\},\{3\},\{4\},\{5\},\{6\},\{7\}\}\\
\text{partition}\,(S,\pi_1) &= \{\{0,6\},\{1,7\},\{2,4\},\{3,5\}\}\\
\text{partition}\,(S,\pi_2) &= \{\{0\},\{1\},\{2,4\},\{3,5\},\{6\},\{7\}\}\\
\text{partition}\,(S,\pi_3) &= \{\{0,6\},\{1,7\},\{2\},\{3\},\{4\},\{5\}\}
\end{aligned} \tag{10.13}$$

The partition of S under the group $\{\pi_0,\pi_1,\pi_2,\pi_3\}$ is now simply the union of the orbits/partitions in (10.12)/(10.13):

$$\begin{aligned}
\text{partition}\,(S,\{\pi_0,\pi_1,\pi_2,\pi_3\}) &= \bigcup_{i\in\{0,1,2,3\}} \text{partition}\,(S,\pi_i)\\
&= \{\{0,6\},\{1,7\},\{2,4\},\{3,5\}\}
\end{aligned} \tag{10.14}$$

10.4. CNF Symmetry

Symmetries of a Boolean function can be classified as being either *semantic* or *syntactic*. Semantic symmetries are intrinsic properties of the function that are independent of any particular representation we may choose for it. In contrast, syntactic symmetries correspond to specific algebraic representations of the function, e.g., an SOP, POS, or some nested expression that might correspond to a circuit representation. Our focus will be on the CNF representation of a function since it is the usual form that is processed by SAT solvers. We should note, however, that a given function can be represented by many equivalent CNF formulas. In particular, some of these formulas are unique representations and can, thus, be used to identify a function's semantic symmetries. Examples of such unique CNF forms include the set of a function's maxterms, the complete set of its prime implicates, or (if it exists) its unique minimal CNF expression. Such representations, however, are generally exponential in the number of variables rendering any attempt to analyze them intractable.

Table 10.6. Symmetries of different CNF formulas for the example function in (10.1). The formula in row 1 consists of the function's maxterms, whereas that in row 2 is the function's complete set of prime implicates. These two forms are unique and will always yield the function's semantic symmetries. Other unique CNF representations, such as a unique minimal CNF form (which in the case of this function is the same as the complete set of prime implicates) will also yield the group of semantic symmetries. The remaining formulas will, generally, yield syntactic symmetries which are subgroups of the group of semantic symmetries.

(Equivalent) CNF Formulas	Symmetries			
	Identity ( )	Variable (a,b)	Value $(a,a')(b,b')$	Mixed (a,b')
1. $(a+b+c)(a+b+c')(a+b'+c)$ $(a'+b'+c)(a'+b'+c')(a'+b+c)$	✓	✓	✓	✓
2. $(a+b)(a'+b')(c)$	✓	✓	✓	✓
3. $(a+b)(a'+b')(a'+c)(a+c)$	✓	✓	✓	✓
4. $(a+b)(a'+b'+c')(c)$	✓	✓		
5. $(a+c)(a'+c)(a'+b'+c')(a+b+c')$	✓		✓	
6. $(a+b)(a'+b')(a+b'+c)(c)$	✓			✓
7. $(a+b)(b'+c)(a'+b'+c)(c)$	✓			

As mentioned in Sec. 10.3.5, we consider the invariance of an n-variable CNF formula $\varphi(X)$ under a) negation, b) permutation, and c) combined negation and permutation of its variables. We will refer to the symmetries that arise under these three types of transformation as *value*, *variable*, and *mixed* symmetries, respectively, and denote the resulting symmetry group by G_φ. Clearly, G_φ is a subgroup of $\mathcal{NP}_n$, i.e., $G_\varphi \leq \mathcal{N}_n \rtimes \mathcal{P}_n$. The action of G_φ on the formula's literals results in a re-ordering of the formula's clauses and of the literals within the clauses while preserving Boolean consistency (10.9).

While it may be obvious, it is useful to demonstrate how different CNF formulas for the same function can have drastically different symmetries, or even no symmetries other than the identity. Table 10.6 gives the symmetries of seven different formulas for our example function (10.1). Clearly, a good *encoding* is necessary if we are to reap the benefits of symmetry detection and breaking.

10.5. Automorphism Group of a Colored Graph

So far we have been concerned with what groups are and what properties they have. Here we introduce one last group, the automorphism group of a colored graph, and sketch the basic procedure for computing it. As we show in Sec. 10.6, the symmetries of a CNF formula can be derived by computing the automorphism group of an associated graph. Efficient algorithms for graph automorphism are, thus, a critical link in the overall symmetry detection and breaking flow.

Definition 10.5.1. (Automorphism Group [McK81]) Given a graph[5] $G = (V,E)$, with vertex set $V = \{1,2,\ldots,n\}$ and edge set E, along with a partition $\pi(V) = \{V_1, V_2, \ldots, V_k\}$ of its vertices, its *automorphism group* $\text{Aut}(G,\pi)$ is $\{\gamma \in$

[5]The use of G and π in this definition to denote *graph* and *partition*, respectively, should not be confused with their earlier use to denote *group* and *permutation*.

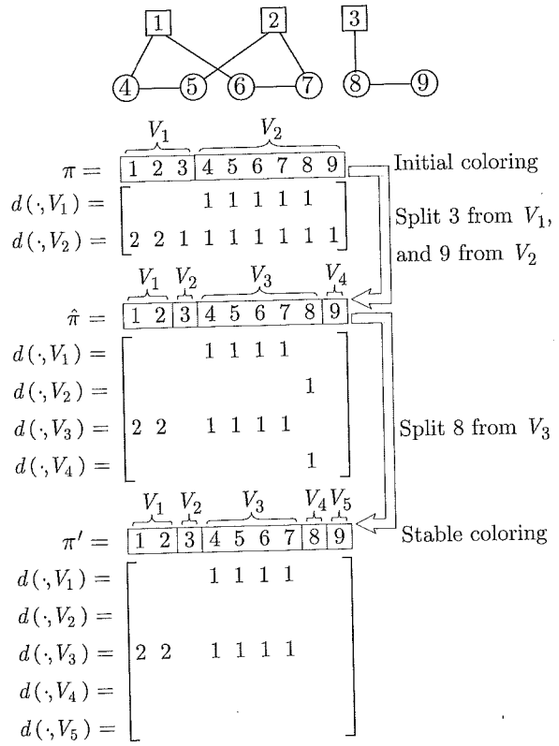


Figure 10.6. Illustration of ordered partition refinement. The "matrix" of vertex degrees is indicated under each coloring (zero entries are shown as blanks, and the degrees of singleton cells are not shown). Efficient implementations of this procedure compute only those vertex degrees necessary to trigger refinement.

$S_n|G^\gamma = G$ and $\pi^\gamma = \pi\}$, where $G^\gamma = (V^\gamma, E^\gamma)$ with $E^\gamma = \{(u^\gamma, v^\gamma)|(u,v) \in E\}$ and $\pi^\gamma = \{V_1^\gamma, \ldots, V_k^\gamma\}$.

In other words, $\mathrm{Aut}(G,\pi)$ is the set of permutations of the graph vertices that map edges to edges and non-edges to non-edges with the restriction that vertices in any given cell of π can only be mapped to vertices in that same cell. It is customary to consider the partition $\pi(V)$ as an assignment of k different colors to the vertices and to view it as a constraint that disallows permutations that map vertices of some color to vertices of a different color. We will be principally interested in *ordered partitions*, also called *colorings*, of the vertices. Specifically, the cells of an ordered partition $\pi(V) = (V_1, V_2, \ldots, V_k)$ form a sequence rather than an unordered set.

The importance of the graph automorphism problem in the context of symmetry detection for CNF formulas stems from the fact that it can be solved fairly efficiently for a large class of graphs. In fact, while this problem is known to

be in the complexity class NP [GJ79], it is still an open question whether it is NP-complete or is in P. A straightforward, but hopelessly impractical, procedure for computing $\mathrm{Aut}(G,\pi)$ is to explicitly enumerate all permutations in S_n and to discard those that are not symmetries of G. Efficient graph automorphism algorithms employ more intelligent implicit enumeration procedures that drastically reduce the number of individual permutations that must be examined. The kernel computation in such algorithms is an *ordered partition refinement* procedure similar to that used for state minimization of finite-state automata [AHU74], and their underlying data structure is a search tree whose nodes represent colorings of the graph vertices.

Definition 10.5.2. (Color-Relative Vertex Degree) Given a coloring $\pi(V) = (V_1, V_2, \ldots, V_k)$ of the graph G and a vertex $v \in V$, let $d(v, V_i)$ be the number of vertices in V_i that are adjacent to v in G. Note that $d(v, V)$ is simply the degree of v in G.

Definition 10.5.3. (Stable Coloring) A coloring π is *stable* if

$$d(u, V_i) = d(v, V_i), \ 1 \leqslant i \leqslant |\pi|$$

for all pairs of vertices $u, v \in V_j$, $1 \leq j \leq |\pi|$.

Ordered partition refinement transforms a given initial coloring π into a final *coarsest* stable coloring $\pi' \leq \pi$ by repeatedly splitting cells containing vertices with different vertex degrees (relative to the current coloring). Fig. 10.6 illustrates this procedure on a sample 2-colored graph with 9 vertices. The initial coloring π consists of two cells corresponding to the graph's two types of vertices (square and round). In the first refinement iteration, vertex 3 is split off from V_1 because its degree relative to V_2 is different from those of the two other vertices in V_1. In other words, vertex 3 can be *distinguished* from vertices 1 and 2 and cannot possibly be mapped to either by a graph symmetry. Similarly, vertex 9 is split off from V_2 yielding the intermediate coloring $\hat{\pi}$. The second refinement iteration splits vertex 8 from V_3 yielding the final stable coloring π'.

If the refinement procedure returns a discrete coloring π', i.e. every cell of the partition is a singleton, then all vertices can be distinguished, implying that G has no symmetries besides the identity. However, if π' is not discrete, then there is some non-singleton cell in π' representing vertices that could not be distinguished based on degree and are, thus, candidates for some symmetry. This is checked by selecting some non-singleton cell T of π', called the *target cell*, and forming $|T|$ descendant colorings from π', each identical to π' except that a distinct $t \in T$ is placed in front of $T - \{t\}$. Each of these colorings is subsequently refined, and further descendant colorings are generated if the refined colorings are not discrete; this process is iterated until discrete colorings are reached. The colorings explored in this fashion form a *search tree* with the discrete colorings at the leaves.

Fig.10.7 illustrates this process for an example 6-vertex graph. The initial coloring has a single cell since all vertices have the same color. Refinement yields a stable coloring with two non-singleton cells. At this point, the first cell is chosen as a target, and we branch by creating three descendant colorings. The process is now repeated (i.e., we refine each derived coloring and branch from it if it has

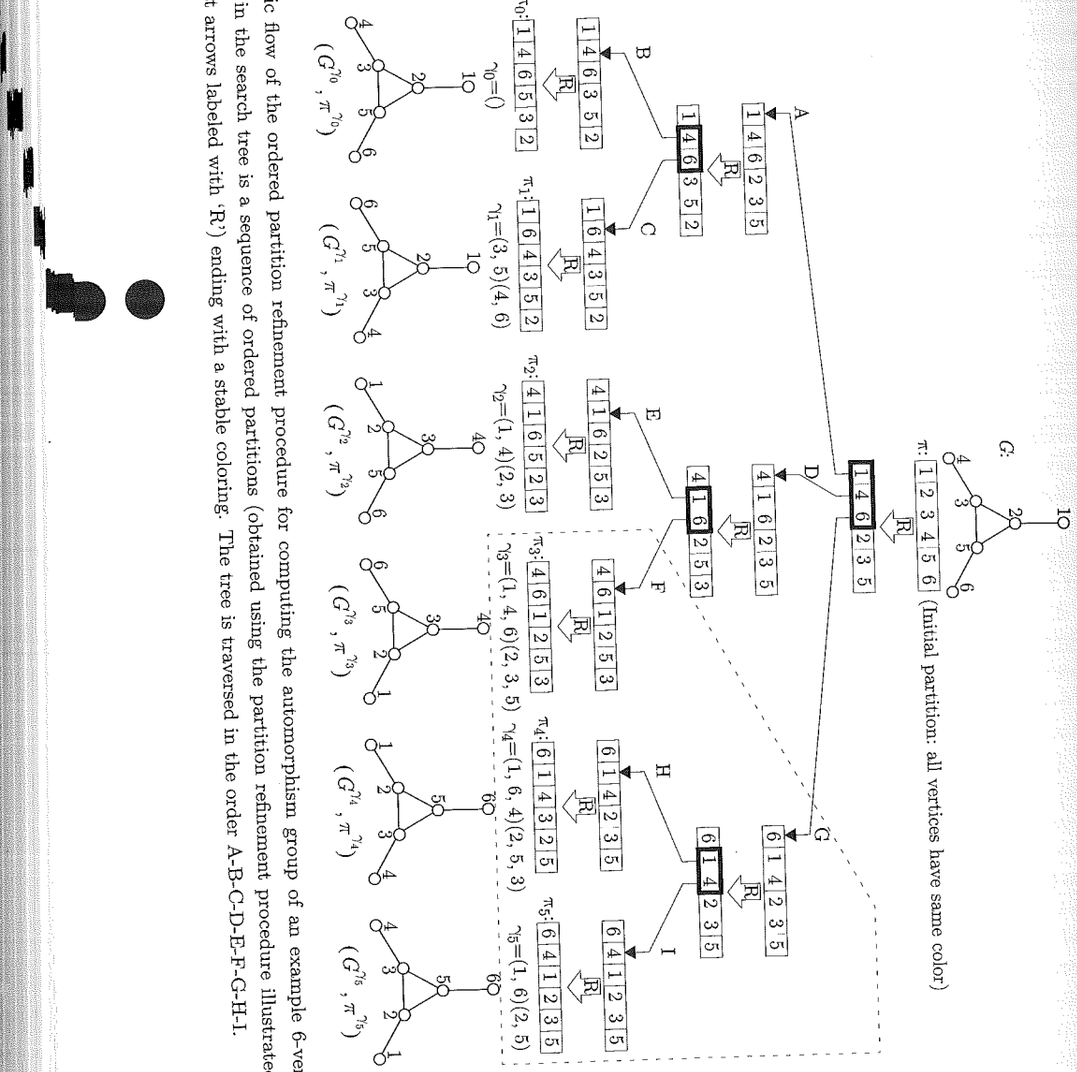


Figure 10.7. Basic flow of the ordered partition refinement procedure for computing the automorphism group of an example 6-vertex single-colored graph. Each node in the search tree is a sequence of ordered partitions (obtained using the partition refinement procedure illustrated in Fig. 10.6 and indicated by the fat arrows labeled with 'R') ending with a stable coloring. The tree is traversed in the order A-B-C-D-E-F-G-H-I.

Table 10.7. Automorphism results on a sample of very large sparse graphs (sorted in descending order by number of vertices) using a modern graph automorphism tool. The *bigblue* and *adaptec* graphs are derived from circuits used in the ISPD 2005 placement competition [ISP]. The *CA* and *FL* graphs represent, respectively, the road networks of California and Florida [U.S]. The *Internet* graph represents the interconnections of the major routers on the Internet's North American backbone [CBB00, GT00]. Note the extreme sparsity of each of these graphs and of their symmetry generators. Note also the astronomical orders of the graphs' automorphism groups. These data are from [DSM08].

Graph $G = (V, E)$				Symmetry Group Aut(G)			Time (s)
Name	$\|V\|$	$\|E\|$	Avg. Deg.	$\|\text{Aut}(G)\|$	Generators	Avg. Supp.	
bigblue4	3,822,980	8,731,076	4.568	10^{119182}	215,132	2.219	5.361
bigblue3	1,876,918	3,812,614	4.063	10^{52859}	98,567	2.129	2.337
CA	1,679,418	2,073,394	2.469	10^{14376}	44,439	2.280	0.838
FL	1,063,465	1,329,206	2.500	10^{13872}	33,225	2.466	0.543
adaptec4	878,800	1,880,109	4.279	10^{24443}	53,857	2.261	0.989
adaptec3	800,506	1,846,242	4.613	10^{15450}	36,289	2.157	0.929
Internet	284,805	428,624	3.010	10^{83687}	108,743	2.281	0.405

non-singleton cells) in a depth-first manner until discrete colorings are reached. In this example, the search tree terminates in six leaves corresponding to six different discrete colorings.

The next step in the process is to derive from these colorings permutations of the graph vertices that correspond to graph symmetries. This is done by choosing one of these colorings as a reference, and computing the permutations that transform it to the other colorings. For the example of Fig.10.7, we chose the left-most coloring π_0 as the reference; the permutations that convert it to the other colorings, including itself, are labeled γ_0 to γ_5. Each such permutation γ_i is now checked to determine if it is a graph symmetry, i.e., if $G^{\gamma_i} = G$. In this example all six permutations are indeed symmetries of the graph, yielding $\text{Aut}(G) = \{\gamma_0, \gamma_1, \gamma_2, \gamma_3, \gamma_4, \gamma_5\}$. A permutation that does not correspond to a symmetry of the graph triggers backtracking in the search tree so that other branches can be explored.

While the above sketch of graph automorphism algorithms is sufficient for a basic understanding of how they operate, we should note that their performance in practice depends critically on aggressive pruning of the search tree to avoid deriving permutations that can be obtained as the product of other permutations that have already been found. For our running example, the portion of the search tree enclosed by the dashed outline (nodes F, G, H, and I) can be safely pruned away since the permutations identified at its leaves can be expressed in terms of permutations γ_1 and γ_2. Ideally, the output of a graph automorphism algorithm should be a set of irredundant generators. In practice, however, the overhead of performing the group-theoretic pruning necessary to guarantee this outcome tends to be excessive. Instead, graph automorphism tools are designed to produce at most $n-1$ generators for an input graph with n vertices, providing an exponentially smaller representation of the complete set of symmetries which is often, but not guaranteed to be, irredundant.

Modern implementations of graph automorphism can process graphs with

millions of vertices in a matter of seconds on commodity personal computers. Key to this remarkable performance is the extreme *sparsity* of both the graphs themselves and the generators of their symmetry groups. In this context, sparsity is taken to mean that the average vertex degree in the graph and the average support of the resulting symmetry generators are both much smaller than the number of graph vertices. Incorporating knowledge of both types of sparsity in the basic automorphism algorithm can result in substantial pruning of the search tree, essentially yielding a tree whose size is linear, rather than quadratic, in the total support of the symmetry generators [DSM08]. Table 10.7 provides empirical evidence of these observations on a number of very large sparse graphs.

10.6. Symmetry Detection

The basic approach for identifying the symmetries of a CNF formula is through reduction to, and solution of, an associated colored graph automorphism problem. Specifically, given a CNF formula consisting of m clauses and l (total) literals over n variables, construct a corresponding colored graph with $m+2n$ nodes and $l+n$ edges as follows:

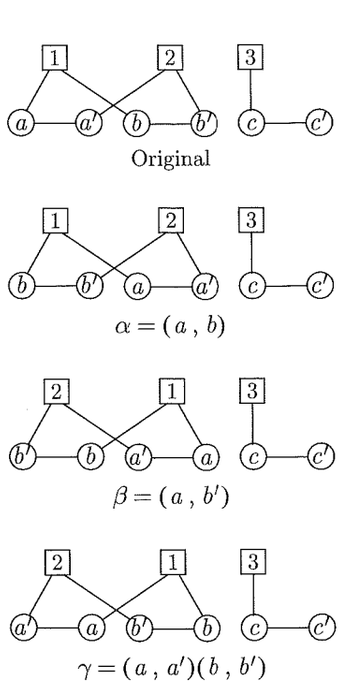
- **Clause nodes**: represent each of the m clauses by a node of color 0
- **Positive literal nodes**: represent each of the n positive literals by a node of color 1
- **Negative literal nodes**: represent each of the n negative literals by a node of color 1
- **Clause edges**: represent each of the m clauses by a set of edges that connect the clause node to the nodes of the literals that appear in the clause
- **Boolean consistency edges**: connect each pair of literal nodes that correspond to the same variable by an edge

The choice of two node colors insures that clause nodes can only be mapped to clause nodes and literal nodes can only be mapped to literal nodes. Using one color for all literal nodes also makes it possible to detect the mixed symmetries of the formula. Pure variable and pure value symmetries can be identified by using more colors for the literal nodes. Specifically, variable symmetries can be identified by using two colors: color 1 for positive literal nodes and color 2 for negative literal nodes. Identifying pure value symmetries requires n colors: color 1 for the literals of the first variable, color 2 for the literals of the second variable, and so on. This construction, thus, is general enough to identify variable, value, and mixed symmetries by a suitable assignment of colors to literal nodes. Fig. 10.8 illustrates these variants, along with the symmetries they yield, for the three-variable CNF formula $(a+b)(a'+b')(c)$ of our example function in (10.1).

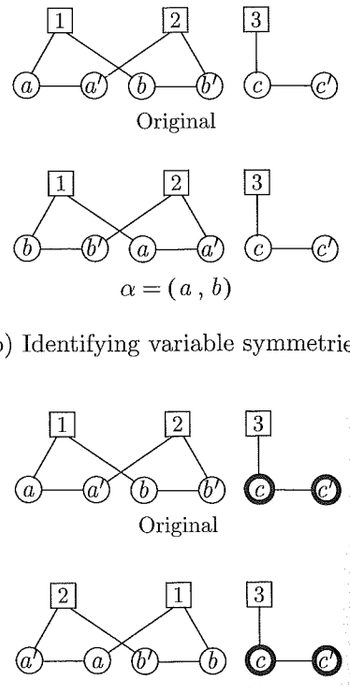
Noting that graph automorphism algorithms tend to be more sensitive to the number of vertices of an input graph than to the number of its edges, the above construction can be optimized to reduce the number of graph vertices while preserving its automorphism group. Specifically, single-literal clauses can be eliminated through Boolean Constraint Propagation (BCP) yielding an equivalent formula that has only clauses with 2 or more literals each. A further reduction in

Table 10.8. Symmetries of a selection of CNF benchmark families along with symmetry detection times in seconds.

Benchmark						Symmetry Group									
		Parameters				Mixed Symmetries				Variable Symmetries			Value Symmetries		
Family	Name	Vars	Clauses Total	Bin	Lits	Nodes	Order	Generators Num	Supp	Time	Order	Generators Num	Supp	Time	Order	Generators Num	Supp	Time
Hole	hole7	56	204	196	448	120	2.03E+08	13	194	0.01	1.00E+00	0	0	0	1.00E+00	0	0	0
	hole8	72	297	288	648	153	1.46E+10	15	254	0.01	1.00E+00	0	0	0.01	1.00E+00	0	0	0
	hole9	90	415	405	900	190	1.32E+12	17	322	0.03	1.00E+00	0	0	0	1.00E+00	0	0	0
	hole10	110	561	550	1210	231	1.45E+14	19	398	0.04	1.00E+00	0	0	0.01	1.00E+00	0	0	0
	hole11	132	738	726	1584	276	1.91E+16	21	482	0.07	1.00E+00	0	0	0.01	1.00E+00	0	0	0
	hole12	156	949	936	2028	325	2.98E+18	23	574	0.11	1.00E+00	0	0	0.01	1.00E+00	0	0	0
ChnlRoute	chnl10_11	220	1122	1100	2420	462	4.20E+28	39	1016	0.23	1.00E+00	0	0	0.01	1.00E+00	0	0	0
	chnl10_12	240	1344	1320	2880	504	6.04E+30	41	1112	0.3	1.00E+00	0	0	0.01	1.00E+00	0	0	0
	chnl10_13	260	1586	1560	3380	546	1.02E+33	43	1208	0.39	1.00E+00	0	0	0.01	1.00E+00	0	0	0
	chnl11_12	264	1476	1452	3168	552	7.31E+32	43	1228	0.38	1.00E+00	0	0	0.02	1.00E+00	0	0	0
	chnl11_13	286	1742	1716	3718	598	1.24E+35	45	1334	0.5	1.00E+00	0	0	0.01	1.00E+00	0	0	0
	chnl11_20	440	4220	4180	8800	920	1.89E+52	59	2076	2.09	1.00E+00	0	0	0.05	1.00E+00	0	0	0
FPGARoute	fpga10_8	120	448	360	1200	328	6.69E+11	22	512	0.04	1.00E+00	0	0	0.01	1.00E+00	0	0	0
	fpga10_9	135	549	450	1485	369	1.50E+13	23	446	0.05	1.00E+00	0	0	0.01	1.00E+00	0	0	0
	fpga12_8	144	560	456	1488	392	2.41E+13	24	624	0.07	1.00E+00	0	0	0	1.00E+00	0	0	0
	fpga12_9	162	684	567	1836	441	5.42E+14	25	546	0.09	1.00E+00	0	0	0	1.00E+00	0	0	0
	fpga12_11	198	968	825	2640	539	1.79E+18	29	678	0.17	1.00E+00	0	0	0	1.00E+00	0	0	0
	fpga12_12	216	1128	972	3096	588	2.57E+20	32	960	0.2	1.00E+00	0	0	0.01	1.00E+00	0	0	0
	fpga13_9	176	759	633	2031	478	3.79E+15	26	598	0.1	1.00E+00	0	0	0.01	1.00E+00	0	0	0
	fpga13_10	195	905	765	2440	530	1.90E+17	28	668	0.15	1.00E+00	0	0	0.01	1.00E+00	0	0	0
	fpga13_12	234	1242	1074	3396	636	9.01E+20	32	812	0.26	1.00E+00	0	0	0.01	1.00E+00	0	0	0
Groute	s3-3-3-1	864	7592	7014	16038	2306	8.71E+09	26	1056	1.72	8.71E+09	26	1056	1.76	1.00E+00	0	0	0.23
	s3-3-3-3	960	9156	8518	19258	2558	6.97E+10	29	1440	2.97	6.97E+10	29	1440	3.27	1.00E+00	0	0	0.31
	s3-3-3-4	912	8356	7748	17612	2432	2.61E+10	27	1200	2.27	2.61E+10	27	1200	2.57	1.00E+00	0	0	0.26
	s3-3-3-8	912	8356	7748	17612	2432	3.48E+10	28	1296	2.36	3.48E+10	28	1296	2.63	1.00E+00	0	0	0.26
	s3-3-3-10	1056	10862	10178	22742	2796	3.48E+10	28	1440	3.89	3.48E+10	28	1440	4.14	1.00E+00	0	0	0.39
Urq	Urq3_5	46	470	2	2912	560	5.37E+08	29	159	0.02	1.00E+00	0	0	0	5.37E+08	29	171	0.03
	Urq4_5	74	694	2	4272	840	8.80E+12	43	329	0.07	1.00E+00	0	0	0	8.80E+12	43	262	0.09
	Urq5_5	121	1210	2	7548	1450	4.72E+21	72	687	0.38	1.00E+00	0	0	0.01	4.72E+21	72	670	0.42
	Urq6_5	180	1756	4	10776	2112	6.49E+32	109	1155	1.2	1.00E+00	0	0	0.01	6.49E+32	109	865	1.31
	Urq7_5	240	2194	2	13196	2672	1.12E+43	143	2035	2.72	1.00E+00	0	0	0.02	1.12E+43	143	1519	2.84
	Urq8_5	327	3252	0	20004	3906	1.61E+60	200	2788	7.04	1.00E+00	0	0	0.04	1.61E+60	200	2720	8.17
XOR	x1_16	46	122	2	364	212	1.31E+05	17	127	0	2.00E+00	1	2	0	6.55E+04	16	163	0
	x1_24	70	186	2	556	324	1.68E+07	24	280	0.01	1.00E+00	0	0	0	1.68E+07	24	278	0.01
	x1_32	94	250	2	748	436	4.29E+09	32	338	0.02	1.00E+00	0	0	0	4.29E+09	32	395	0.02
	x1_36	106	282	2	844	492	6.87E+10	36	462	0.03	1.00E+00	0	0	0	6.87E+10	36	543	0.02
Pipe	2pipe_1_ooo	834	7026	4843	19768	3851	8.00E+00	3	724	0.56	2.00E+00	1	634	0.51	1.00E+00	0	0	0.17
	2pipe_2_ooo	925	8213	5930	23161	4133	3.20E+01	5	888	0.76	2.00E+00	1	708	0.66	1.00E+00	0	0	0.22
	2pipe	861	6695	5796	18637	2621	1.28E+02	7	880	0.62	8.00E+00	3	700	0.52	1.00E+00	0	0	0.18



(a) Identifying mixed symmetries



(b) Identifying variable symmetries

(c) Identifying value symmetries

Figure 10.8. Colored graph constructions and resultant symmetries for the example formula $(a + b)(a' + b')(c)$. (a) Using two colors (square nodes: color 0; circle nodes: color 1) yields the symmetry group $G = \{(), (a, b), (a, b'), (a, a')(b, b')\}$. (b) Using three colors (shaded circles: color 2) yields the symmetry group $G = \{(), (a, b)\}$. (c) Using four colors (bolded circles: color 3) yields the symmetry group $G = \{(), (a, a')(b, b')\}$.

the number of graph vertices is achieved by modeling binary clauses (i.e., 2-literal clauses) using graph edges instead of graph vertices. As we demonstrate shortly, the CNF formulas used to model problems from a variety of application domains tend to have a large fraction of binary clauses. Thus, this optimization is particularly significant for the overall efficiency of symmetry detection. Care must be taken, however, when applying this optimization since it may, on rare occasions, lead to spurious symmetries that result from unintended mappings between binary clause edges and Boolean consistency edges. Such symmetries violate the Boolean consistency requirement (10.9) and, as shown in [ARMS03], arise when the CNF formula contains chains of implications such as $(x'_1 + x_2)(x'_2 + x_3) \cdots (x'_k + x_1)$. The set of generators returned by a graph automorphism program must, thus, be checked for members that violate (10.9) before reporting the results back. Several options of how to filter these spurious symmetries are described in [ARMS03].

The results of applying the above symmetry detection procedure to a selection of CNF instances are given in Table 10.8. These instances are drawn from the following six benchmark families.

- **Hole**: This is the family of unsatisfiable pigeon-hole instances. Instance 'hol*en*' corresponds to placing $n + 1$ pigeons in n holes [DIM].
- Difficult CNF instances that represent the routing of wires in integrated circuits [ARMS02, ARMS03]. This family includes three members: the first two share the characteristic of the pigeon-hole family (namely placing n objects in m slots such that each object is in a slot and no two objects are in the same slot) whereas the third member is based on the notion of *randomized flooding*:
  - **ChnlRoute**: This is a family of unsatisfiable channel routing instances. Instance 'chnl*n_m*' models the routing of m wires through n tracks in a routing channel.
  - **FPGARoute**: This is a family of satisfiable instances that model the routing of wires in the channels of field-programmable integrated circuits [NASR01]. Instance 'fpga*n_m*' models the routing of m wires in channels of n tracks.
  - **GRoute**: This is a family of randomly-generated satisfiable instances that model the routing of global wires in integrated circuits. Instance 's*n-m-c-k*' corresponds to global routing of wires on an n-by-m grid with *edge capacity* between grid cells equal to c tracks. The last integer k in the instance name serves as the instance identifier.
- **Urq**: The is the family of unsatisfiable randomized instances based on expander graphs [Urq87].
- **XOR**: This is a family of various *exclusive or* chains from the SAT'02 competition [SAT].
- **Pipe**: This is a family of difficult unsatisfiable instances that model the functional correctness requirements of modern out-of-order microprocessor CPUs [VB01].

The results in Table 10.8 were generated by a) constructing three colored graphs for each CNF instance and, b) using the **saucy** graph automorphism program [DLSM04] to obtain corresponding sets of generators for the graphs' automorphism groups. These generators were then postprocessed (by dropping cycles that permute clause vertices, and by relabeling negative literal vertices to relate them back to their positive literals) to obtain the symmetry generators of the CNF instance. The generator sets corresponding to the three graph variants identify, respectively, a formula's mixed, variable, and value symmetries. The experiments were conducted on a 3GHz Intel Xeon workstation with 4GB RAM running the Redhat Linux Enterprise operating system.
For each benchmark instance, the table lists the following data:

- Instance parameters: family, name, number of variables, total number of clauses, number of binary clauses, number of literals, and number of nodes in the corresponding colored graph.
- Symmetry group characteristics for each of the three variants of the colored graph: group order, number of generators along with their total support,

and symmetry detection time.

Examination of the data in Table 10.8 leads to the following general ob

- Except for the Pipe microprocessor verification benchmarks have a handful of symmetries, the number of symmetries in thes is extremely large.
- Symmetry detection is very fast, in most cases taking only ti of a second. The notable exceptions are the GRoute benchma larger Urq benchmarks which require on the order of a few se is partly explained by the large graphs used to model them w those of the other instances, have thousands rather than hundre On the other hand, the fast detection times for the Pipe instan their large graphs, can be attributed to the small number of sy those graphs.
- While **saucy** guarantees that the maximum number of generat puts will be no more than the number of graph nodes minus one, in all cases the number of produced generators is much less. In fact, the upper bound given in Theorem 10.3.3 for the size of an irredu generators is reached only for the Urq and Pipe benchmarks. of generators in other cases is less than this bound, sometime factor.
- The Hole and wire routing benchmarks exhibit only pure va metries. This is to be expected as these benchmarks essenti some sort of capacity constraint on a set of interchangeable var also that the two varieties of colored graphs corresponding to variable symmetries yield the same number of generators in al
- The Urq and XOR benchmarks, except for x1_16, exhibit onl symmetries. Again the graphs for mixed and value symmetr same number of generators in those cases. But note that th these generators is different suggesting that **saucy** is produc sets of mixed and value symmetry generators. The x1_16 ins only one in the entire set of benchmarks in the table to have symmetry: 2 variable permutations, 6.55E+4 variable negatio composition.
- The Pipe benchmarks do not have any value symmetries, but able symmetries and mixed symmetries that involve simultane permutations and negations.
- Finally, the support of the symmetry generators in all cases fraction of the total number of variables, ranging from 4% to mean of 13%. In other words, the generators are extremely sp

We should note that these benchmarks are relatively small and th
conceivable that other benchmark families might yield a different s
However, it seems safe to assume that the detection of the symmetrie
mulas through reduction to graph automorphism is computationally
using today's graph automorphism programs, and that the use of su
as pre-processors for SAT solvers can be a viable approach in many

Table 10.9. Symmetries of the formula $\varphi(a,b,c) = (a+b'+c')(a+b+c)(a'+b+c')$ along with their orbits and lex-leader predicates (the lex-leaders in each orbit are shown in bold.) Note that we have omitted the identity permutation π_0 since each of the 8 truth assignments will be in its own orbit yielding a lex-leader predicate which is the constant 1 function.

Permutation	Orbits	Lex-Leader Predicate
$\pi_1 = (a,b)$	$\{\mathbf{0} \mapsto 0, \mathbf{1} \mapsto 1, \mathbf{2} \mapsto 4 \mapsto 2, \mathbf{3} \mapsto 5 \mapsto 3, \mathbf{6} \mapsto 6, \mathbf{7} \mapsto 7\}$	$\sum_{(a,b,c)} (0,1,2,3,6,7) = (a'+b)$
$\pi_2 = (a,c')$	$\{\mathbf{0} \mapsto 5 \mapsto 0, \mathbf{1} \mapsto 1, \mathbf{2} \mapsto 7 \mapsto 2, \mathbf{3} \mapsto 3, \mathbf{4} \mapsto 4, \mathbf{6} \mapsto 6\}$	$\sum_{(a,b,c)} (0,1,2,3,4,6) = (a'+c')$
$\pi_3 = (b,c')$	$\{\mathbf{0} \mapsto 3 \mapsto 0, \mathbf{1} \mapsto 1, \mathbf{2} \mapsto 2, \mathbf{4} \mapsto 7 \mapsto 4, \mathbf{5} \mapsto 5, \mathbf{6} \mapsto 6\}$	$\sum_{(a,b,c)} (0,1,2,4,5,6) = (b'+c')$
$\pi_4 = (a,b,c')$	$\{\mathbf{0} \mapsto \mathbf{3} \mapsto 5 \mapsto 0, \mathbf{1} \mapsto 1, \mathbf{2} \mapsto 7 \mapsto 4 \mapsto 2, \mathbf{6} \mapsto 6\}$	$\sum_{(a,b,c)} (0,1,2,3,6) = (a'+b)(a'+c')$
$\pi_5 = (a,c',b)$	$\{\mathbf{0} \mapsto 5 \mapsto 3 \mapsto 0, \mathbf{1} \mapsto 1, \mathbf{2} \mapsto \mathbf{4} \mapsto 7 \mapsto 2, \mathbf{6} \mapsto 6\}$	$\sum_{(a,b,c)} (0,1,2,4,6) = (a'+c')(b'+c')$

Group	Equivalence Classes	Lex-Leader Predicate
$\{\pi_0, \pi_1, \pi_2, \pi_3, \pi_4, \pi_5\}$	$\{\{\mathbf{0},3,5\},\{\mathbf{1}\},\{\mathbf{2},4,7\},\{\mathbf{6}\}\}$	$\sum_{(a,b,c)} (0,1,2,6) = (a'+b)(b'+c')$

10.7. Symmetry Breaking

Given a CNF formula $\varphi(X)$ along with its group of symmetries G_φ the goal now is to augment $\varphi(X)$ with additional predicates that break all of its symmetries. Recall that G_φ induces an equivalence relation on the set of truth assignments, i.e., it partitions the space of possible assignments into equivalence classes. Complete symmetry breaking amounts to constructing predicates that select exactly one representative assignment from each equivalence class. The most common approach for accomplishing this objective is to order the assignments numerically, and to use the leq predicate introduced in Sec. 10.2 to choose the least assignment in each equivalence class. Specifically, let $\mathrm{PP}(\pi; X)$ denote the *permutation predicate* of permutation $\pi \in G_\varphi$ defined as:

$$\begin{aligned} \mathrm{PP}(\pi; X) &= \mathrm{leq}(X, X^\pi) \\ &= \bigwedge_{i \in [0,n-1]} \left[\left[\bigwedge_{j \in [i+1,n-1]} \left(x_j = x_j^\pi \right) \right] \rightarrow \left(x_i \leqslant x_i^\pi \right) \right] \end{aligned} \tag{10.15}$$

$\mathrm{PP}(\pi; X)$ is a Boolean function that evaluates to 1 for each assignment X^* such that $X^* \leq \pi(X^*)$ and evaluates to 0 otherwise. More specifically, let $X_0^* \mapsto X_1^* \mapsto \cdots \mapsto X_{k-1}^* \mapsto X_0^*$ be an orbit of π, i.e. π maps each assignment X_i^* to the next assignment $X_{(i+1) \mod k}^*$; then $\mathrm{PP}(\pi; X)$ is true for only those assignments that are numerically smaller than their successor assignments as we traverse the orbit. In particular, $\mathrm{PP}(\pi; X)$ will be true for the smallest assignment in the orbit. We will refer to $\mathrm{PP}(\pi; X)$ as a *lex-leader predicate* for permutation π. We can now construct a Boolean predicate that breaks *every* symmetry in G_φ by conjoining all of its permutation predicates:

$$\rho(G_\varphi; X) = \bigwedge_{\pi \in G_\varphi} \mathrm{PP}(\pi; X) \tag{10.16}$$

This symmetry-breaking predicate (SBP) will be true for exactly the least assignment in each equivalence class induced by G_φ on the set of truth assignments. An example illustrating the construction of this predicate is shown in Table 10.9.

Using the SBP in (10.16), we can now determine the satisfiability of $\varphi(X)$ by invoking a SAT solver on the formula

$$\psi(X) \equiv \varphi(X) \wedge \rho(G_\varphi; X) \tag{10.17}$$

Clearly, if $\varphi(X)$ is unsatisfiable, then so is $\psi(X)$. On the other hand, if $\varphi(X)$ is satisfiable then its restriction by $\rho(G_\varphi; X)$ preserves only those solutions that are *numerically least* in their orbits. Either way, we can determine the satisfiability of $\varphi(X)$ by, instead, checking the satisfiability of $\psi(X)$.

To apply a SAT solver to (10.17), the permutation predicates in (10.15) must first be converted to CNF. A simple way to accomplish this is to introduce n auxiliary *equality* variables

$$e_i \equiv (x_i = x_i^\pi), \quad 0 \leqslant i \leqslant n-1 \tag{10.18}$$

allowing us to re-write (10.15) as

$$\mathrm{PP}(\pi; X) = \bigwedge_{i \in [0,n-1]} \left(e'_{i+1} + e'_{i+2} + \cdots + e'_{n-1} + x'_i + x_i^\pi \right) \tag{10.19}$$

This leads to a CNF representation of the permutation predicate whose size is

$$\begin{aligned} \mathrm{clauses}(\mathrm{PP}(\pi; X)) &= 5n \\ \mathrm{literals}(\mathrm{PP}(\pi; X)) &= 0.5\left(n^2 + 27n\right) \end{aligned} \tag{10.20}$$

Table 10.10. Comparison of CNF formula size and permutation predicate size from (10.20)

	Benchmark				Permutation Predicate		
Family	Name	Vars	Clauses	Lits	Vars	Clauses	Lits
Hole	hole7	56	204	448	56	280	2324
	hole8	72	297	648	72	360	3564
	hole9	90	415	900	90	450	5265
	hole10	110	561	1210	110	550	7535
	hole11	132	738	1584	132	660	10494
	hole12	156	949	2028	156	780	14274
ChnlRoute	chnl10_11	220	1122	2420	220	1100	27170
	chnl10_12	240	1344	2880	240	1200	32040
	chnl10_13	260	1586	3380	260	1300	37310
	chnl11_12	264	1476	3168	264	1320	38412
	chnl11_13	286	1742	3718	286	1430	44759
	chnl11_20	440	4220	8800	440	2200	102740
FPGARoute	fpga10_8	120	448	1200	120	600	8820
	fpga10_9	135	549	1485	135	675	10935
	fpga12_8	144	560	1488	144	720	12312
	fpga12_9	162	684	1836	162	810	15309
	fpga12_11	198	968	2640	198	990	22275
	fpga12_12	216	1128	3096	216	1080	26244
	fpga13_9	176	759	2031	176	880	17864
	fpga13_10	195	905	2440	195	975	21645
	fpga13_12	234	1242	3396	234	1170	30537
Groute	s3-3-3-1	864	7592	16038	864	4320	384912
	s3-3-3-3	960	9156	19258	960	4800	473760
	s3-3-3-4	912	8356	17612	912	4560	428184
	s3-3-3-8	912	8356	17612	912	4560	428184
	s3-3-3-10	1056	10862	22742	1056	5280	571824
Urq	Urq3_5	46	470	2912	46	230	1679
	Urq4_5	74	694	4272	74	370	3737
	Urq5_5	121	1210	7548	121	605	8954
	Urq6_5	180	1756	10776	180	900	18630
	Urq7_5	240	2194	13196	240	1200	32040
	Urq8_5	327	3252	20004	327	1635	57879
XOR	x1_16	46	122	364	46	230	1679
	x1_24	70	186	556	70	350	3395
	x1_32	94	250	748	94	470	5687
	x1_36	106	282	844	106	530	7049
Pipe	2pipe_1_ooo	834	7026	19768	834	4170	359037
	2pipe_2_ooo	925	8213	23161	925	4625	440300
	2pipe	861	6695	18637	861	4305	382284

The size of a single permutation predicate can, thus, significantly exceed the size of the original formula! Table 10.10 compares the size of each benchmark in Table 10.8 to the size of a single permutation predicate as formulated in (10.18) and (10.19). In practice, this dramatic increase in the size of the formula negates much of the hoped-for reduction in the search space due to symmetry breaking. This is further compounded by the fact that full symmetry breaking requires adding, according to (10.16), a quadratically-sized predicate *for each permutation in the symmetry group*. As the data in Table 10.8 clearly demonstrate, the orders of the symmetry groups for most benchmarks are exponential in the number of variables, rendering direct application of (10.18)-(10.19) infeasible.

This analysis may suggest that we have reached an impasse: the theoretical possibility of significantly pruning the search space by breaking a formula's symmetries seems to be at odds with the need to add an exponentially-sized SBP to the formula causing a SAT solver to grind to a halt. We show next a practical approach to resolve this impasse and demonstrate its effectiveness on the benchmark families analyzed in Sec. 10.6. The approach rests on two key ideas: a) reducing the size of the permutation predicate so that it becomes sub-linear, rather than quadratic, in the number of variables, and b) relaxing the requirement to break all symmetries and opting, instead, to break "enough" symmetries to reduce SAT search time.

10.7.1. Efficient Formulation of the Permutation Predicate

To obtain an efficient CNF representation of the permutation predicate in (10.15) it is necessary to introduce some additional notation that facilitates the manipulation of various subsets of the variables $\{x_i | 0 \leq i \leq n-1\}$. Let I_n denote the integer set $\{0, 1, \ldots, n-1\}$, and let upper-case "index variables" I and J denote non-empty subsets of I_n as appropriate. Given an index set I and an index $i \in I$, define the *index selector* functions:

$$\begin{aligned} \text{pred}(i, I) &= \{j \in I \,|\, j < i\} \\ \text{succ}(i, I) &= \{j \in I \,|\, j > i\} \end{aligned} \tag{10.21}$$

A permutation is said to be a *phase-shift* permutation if it contains one or more 2-cycles that have the form (x_i, x_i'), i.e., cycles that map a variable to its complement. Such cycles will be referred to as *phase-shift cycles*. Given a permutation π, we now define

$$\text{ends}(\pi) = \{i \in I_n |\ i \text{ is the smallest index of a literal in a non-phase-shift cycle of } \pi\} \tag{10.22}$$

and

$$\text{phase-shift}(\pi) = \max\{i \in I_n |\, x_i^\pi = x_i'\} \tag{10.23}$$

In other words, based on the assumed total ordering $x_{n-1} \succ x_{n-2} \succ \cdots \succ x_0$, ends($\pi$) identifies the *last* literal in each non-phase-shift cycle of π whereas phase-shift(π) identifies the literal of the *first* phase-shift cycle.

We begin the simplification of the permutation predicate in (10.15) by rewriting it as a conjunction of *bit predicates*:

$$\text{BP}(\pi; x_i) = \left[\bigwedge_{j \in \text{succ}(i, I_n)} (x_j = x_j^\pi)\right] \rightarrow (x_i \leqslant x_i^\pi) \tag{10.24}$$

$$\text{PP}(\pi; X) = \bigwedge_{i \in I_n} \text{BP}(\pi, x_i) \tag{10.25}$$

Minimization of the CNF representation of the permutation predicate is based on two key optimizations. The first utilizes the *cycle structure* of a permutation to eliminate tautologous bit predicates and is accomplished by replacing I_n in (10.24) and (10.25) with $I \subset I_n$ where $|I| \ll n$. This optimization can be viewed as replacing n in (10.20) by a much smaller number m. The second optimization decomposes the multi-way conjunction in (10.24) into a chain of 2-way conjunctions to yield a CNF formula whose size is linear, rather than quadratic, in m. Fig. 10.9 provides an example illustrating these optimizations.

Elimination of redundant BPs. Careful analysis of (10.24) reveals three cases in which a BP is tautologous, and hence redundant. The first corresponds to bits that are mapped to themselves by the permutation, i.e., $x_i^\pi = x_i$. This makes the consequent of the implication in (10.24), and hence the whole bit predicate, unconditionally true. With a slight abuse of notation, removal of such BPs is easily accomplished by replacing the index set I_n in (10.24) and (10.25) with supp(π)[6]. For sparse permutations, i.e., permutations for which $|\text{supp}(\pi)| \ll n$, this change alone can account for most of the reduction in the CNF size of the PP.

The second case corresponds to the BP of the last bit in each non-phase-shift cycle of π. "Last" here refers to the assumed total ordering on the variables. Assume a cycle involving the variables $\{x_j | j \in J\}$ for some index set J and let $e = \min(J)$. Such a cycle can always be written as $(x_s, \ldots, x_e)$ where $s \in J$. Using equality propagation, the portion of the antecedent in (10.24) involving the variables of this cycle can, thus, be simplified to:

$$\left[\bigwedge_{j \in J - \{e\}} (x_j = x_j^\pi)\right] = (x_s = x_e)$$

and since $(x_s - x_e) \rightarrow (x_e \leq x_s)$ is a tautology, the whole bit predicate becomes tautologous. Elimination of these BPs is accomplished by a further restriction of the index set in (10.24) and (10.25) to just supp(π) – ends(π) and corresponds to a reduction in the number of BPs from n to $m = |\text{supp}(\pi)| - \text{cycles}(\pi)$ where cycles(π) is the number of non-phase-shift cycles of π.

The third and last case corresponds to the BPs of those bits that occur after the first "phase-shifted variable." Let i be the index of the first variable for which $x_i^\pi = x_i'$. Thus, $e_i = 0$ and all BPs for $j < i$ have the form $0 \rightarrow (x_j \leq x_j^\pi)$ making them unconditionally true.

[6] Technically, we should replace I_n with $\{i \in I_n | \dot{x}_i \in \text{supp}(\pi)\}$.

$$\pi = \begin{pmatrix} x_9 & x_8 & x_7 & x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 \\ x_6 & x_8 & x_2 & x_9 & x_5' & x_7' & x_3 & x_4' & x_1 & x_0 \end{pmatrix} = (x_9, x_6)(x_7, x_2, x_4')(x_5, x_5')$$

$\mathrm{supp}(\pi) = \{2, 4, 5, 6, 7, 9\}$, phase-shift$(\pi) = 5$, ends$(\pi) = \{2, 6\}$

$\mathrm{pred}(\text{phase-shift}(\pi), I_{10}) = \{5, 7, 9\}$

$I = \mathrm{supp}(\pi) - \mathrm{ends}(\pi) - \mathrm{pred}(\text{phase-shift}(\pi), I_{10}) = \{4\}$

(a) Permutation in tabular and cycle notation, along with various associated index sets.

$\mathrm{BP}(\pi, x_9) = (x_9 \leq x_6)$

$\boxed{\mathrm{BP}(\pi, x_8) = (x_9 = x_6) \rightarrow (x_8 \leq x_8')}$

$\mathrm{BP}(\pi, x_7) = (x_9 = x_6)(x_8 = x_8') \rightarrow (x_7 \leq x_2')$

$\boxed{\mathrm{BP}(\pi, x_6) = (x_9 = x_6)(x_8 = x_8')(x_7 = x_2') \rightarrow (x_6 \leq x_9)}$

$\mathrm{BP}(\pi, x_5) = (x_9 = x_6)(x_8 = x_8')(x_7 = x_2')(x_6 = x_9) \rightarrow (x_5 \leq x_5')$

$\mathrm{BP}(\pi, x_4) = (x_9 = x_6)(x_8 = x_8')(x_7 = x_2')(x_6 = x_9)(x_5 = x_5') \rightarrow (x_4 \leq x_7')$

$\mathrm{BP}(\pi, x_3) = (x_9 = x_6)(x_8 = x_8')(x_7 = x_2')(x_6 = x_9)(x_5 = x_5')(x_4 = x_7') \rightarrow (x_3 \leq x_3)$

$\boxed{\mathrm{BP}(\pi, x_2) = (x_9 = x_6)(x_8 = x_8')(x_7 = x_2')(x_6 = x_9)(x_5 = x_5')(x_4 = x_7')(x_3 = x_3) \rightarrow (x_2 \leq x_4')}$

$\boxed{\mathrm{BP}(\pi, x_1) = (x_9 = x_6)(x_8 = x_8')(x_7 = x_2')(x_6 = x_9)(x_5 = x_5')(x_4 = x_7')(x_3 = x_3)(x_2 = x_4') \rightarrow (x_1 \leq x_1)}$

$\boxed{\mathrm{BP}(\pi, x_0) = (x_9 = x_6)(x_8 = x_8')(x_7 = x_2')(x_6 = x_9)(x_5 = x_5')(x_4 = x_7')(x_3 = x_3)(x_2 = x_4')(x_1 = x_1) \rightarrow (x_0 \leq x_0)}$

(b) Permutation's bit predicates. BPs enclosed in boxes with square corners are tautologous because π maps the corresponding bits to themselves. BPs enclosed in boxes with rounded corners are tautologous because they correspond to cycle "ends." The BP's for bits 4 to 0 are tautologous because π maps bit 5 to its complement.

$$\begin{aligned} \mathrm{PP}^{\star}(\pi; X) &= (l_9)(p_9 \rightarrow l_7)(p_7 \rightarrow l_5)(g_9 \rightarrow p_9)(p_9 g_7 \rightarrow p_7) \\ &= (x_9' + x_6)(p_9' + x_7' + x_5)(p_7' + x_5')(x_6' + p_9)(x_6 + p_9)(p_9' + x_7' + p_7)(p_9' + x_5 + p_7) \end{aligned}$$

(c) Linear formulation of the (relaxed) permutation predicate using the chaining 'p' variables.

Figure 10.9. Illustration of the formulation of the permutation predicate according to (10.26) and (10.32).

Taken together, the redundant BPs corresponding to these three cases can be easily eliminated by setting the index set in (10.24) and (10.25) to:

$$I = \mathrm{supp}(\pi) - \mathrm{ends}(\pi) - \mathrm{pred}(\text{phase-shift}(\pi), I_n) \tag{10.26}$$

In the sequel we will refer to the bits in the above index set as "irredundant bits" and use $I = \{i_1, i_2, \ldots, i_m\}$, with $i_1 > i_2 > \cdots > i_m$, to label them. Note that the presence of a phase-shifted variable early in the total order can lead to a drastic reduction in the number of irredundant bits. For example, if $\pi = (x_{n-1}, x'_{n-1}) \cdots$ then $\mathrm{PP}(\pi; X)$ is simply (x'_{n-1}) regardless of how many other variables are moved by π.

Linear construction of PPs through chaining. Elimination of the redundant bit predicates yields the following reduced expression for $\mathrm{PP}(\pi; X)$:

$$\mathrm{PP}(\pi; X) = \bigwedge_{1 \leqslant k \leqslant m} \left\{ \left[\bigwedge_{1 \leqslant j \leqslant k-1} \left(x_{i_j}^{\pi} = x_{i_j} \right) \right] \rightarrow \left(x_{i_k}^{\pi} \leqslant x_{i_k} \right) \right\} \tag{10.27}$$

where the outer conjunction is now over the irredundant index set $\{i_1, \ldots, i_m\}$. We show next how to convert (10.27) to a CNF formula whose size is linear in m. The first step in this conversion is to eliminate the equalities in the inner conjunction. Specifically, we introduce the "ordering" predicates $l_{i_j} = (x_{i_j} \leq x_{i_j}^{\pi})$ and $g_{i_j} = (x_{i_j} \geq x_{i_j}^{\pi})$ to re-write (10.27) as:

$$\mathrm{PP}(\pi; X) = \bigwedge_{1 \leqslant k \leqslant m} \left\{ \left[\bigwedge_{1 \leqslant j \leqslant k-1} l_{i_j} g_{i_j} \right] \rightarrow l_{i_k} \right\} \tag{10.28}$$

The next step utilizes the following easily-proved lemma:

Lemma 10.7.1. (Elimination of Redundant Premise)

$$(a \rightarrow b) \wedge \bigwedge_{i \in I} (abc_i \rightarrow d_i) = (a \rightarrow b) \wedge \bigwedge_{i \in I} (ac_i \rightarrow d_i)$$

Repeated application of this lemma to (10.28) leads to the successive elimination of the "less-than-or-equal" predicates in the inner conjunction:

$$\begin{aligned} &\mathrm{PP}(\pi; X) \\ &= (1 \rightarrow l_{i_1})(l_{i_1} g_{i_1} \rightarrow l_{i_2})(l_{i_1} g_{i_1} l_{i_2} g_{i_2} \rightarrow l_{i_3}) \cdots (l_{i_1} g_{i_1} \cdots l_{i_{m-1}} g_{i_{m-1}} \rightarrow l_{i_m}) \\ &= (1 \rightarrow l_{i_1})(g_{i_1} \rightarrow l_{i_2})(g_{i_1} l_{i_2} g_{i_2} \rightarrow l_{i_3}) \cdots (g_{i_1} l_{i_2} g_{i_2} \cdots l_{i_{m-1}} g_{i_{m-1}} \rightarrow l_{i_m}) \\ &= (1 \rightarrow l_{i_1})(g_{i_1} \rightarrow l_{i_2})(g_{i_1} g_{i_2} \rightarrow l_{i_3}) \cdots (g_{i_1} g_{i_2} \cdots l_{i_{m-1}} g_{i_{m-1}} \rightarrow l_{i_m}) \qquad (10.29) \\ &= \cdots \\ &= (1 \rightarrow l_{i_1})(g_{i_1} \rightarrow l_{i_2})(g_{i_1} g_{i_2} \rightarrow l_{i_3}) \cdots (g_{i_1} g_{i_2} \cdots g_{i_{m-1}} \rightarrow l_{i_m}) \\ &= \bigwedge_{1 \leqslant k \leqslant m} \left\{ \left[\bigwedge_{1 \leqslant j \leqslant k-1} g_{i_j} \right] \rightarrow l_{i_k} \right\} \end{aligned}$$

Next, we decompose the multi-way conjunctions involving the "greater-than-or-equal" predicates by introducing m auxiliary *chaining* variables[7] defined by:

$$\begin{aligned} & p_{i_0} = 1 \\ & p_{i_1} = p_{i_0} \wedge g_{i_1} = g_{i_1} \\ & p_{i_2} = p_{i_1} \wedge g_{i_2} = g_{i_1} \wedge g_{i_2} \\ & \ldots \\ & p_{i_{m-1}} = p_{i_{m-2}} \wedge g_{i_{m-1}} = \bigwedge_{1 \leqslant j \leqslant m-1} g_{i_j} \end{aligned} \tag{10.30}$$

Substituting these definitions, along with those of the ordering predicates, yields the following final expression for the permutation predicate

$$\begin{aligned} \mathrm{PP}(\pi; X) = & \left[\bigwedge_{1 \leqslant k \leqslant m} \left(p_{i_{k-1}} \rightarrow \left(x_{i_k} \leqslant x^{\pi}_{i_k} \right) \right) \right] \wedge \\ & \left[\bigwedge_{1 \leqslant k \leqslant m-1} \left(p_{i_k} = p_{i_{k-1}} \wedge \left(x_{i_k} \geqslant x^{\pi}_{i_k} \right) \right) \right] \end{aligned} \tag{10.31}$$

which can be readily converted to a CNF formula consisting of, approximately, $4m$ 3-literal clauses and m 2-literal clauses for a total of $14m$ literals. A further reduction is possible by replacing the equalities in the second conjunction in (10.31) with one-way implications, effectively relaxing the permutation predicate to

$$\begin{aligned} \mathrm{PP}^*(\pi; X) = & \left[\bigwedge_{1 \leqslant k \leqslant m} \left(p_{i_{k-1}} \rightarrow \left(x_{i_k} \leqslant x^{\pi}_{i_k} \right) \right) \right] \wedge \\ & \left[\bigwedge_{1 \leqslant k \leqslant m-1} \left(p_{i_{k-1}} \wedge \left(x_{i_k} \geqslant x^{\pi}_{i_k} \right) \rightarrow p_{i_k} \right) \right] \end{aligned} \tag{10.32}$$

Since $\mathrm{PP}(\pi; X) \leq \mathrm{PP}^*(\pi; X)$, it is now possible for $\mathrm{PP}^*(\pi; X)$ to have solutions that violate $\mathrm{PP}(\pi; X)$. These solutions follow the pattern:

$p_{i_{k-1}}$	g_{i_k}	p_{i_k}
0	0	1
0	1	1
1	0	1

In other words, p_{i_k} is 1 when either $p_{i_{k-1}}$ or g_{i_k} is 0. Each such solution, however, can be modified to a solution that does satisfy $\mathrm{PP}(\pi; X)$ by simply changing the value of p_{i_k} from 1 to 0. We can thus use $\mathrm{PP}^*(\pi; X)$ without compromising correctness; this predicate consists of $3m$ 3-literal clauses for a total of $9m$ literals.

10.7.2. Partial Symmetry Breaking

By filtering all but the least assignment in each orbit of the symmetry group G_φ, the SBP in (10.16) is able to break every symmetry of the associated CNF formula

[7] Actually, we only need $m - 1$ auxiliary variables since p_{i_0} stands for the constant 1.

$\varphi(X)$. For exponentially large symmetry groups, however, such *full symmetry breaking* is infeasible. Even when it is possible to construct a full SBP, its ability to prune away symmetric assignments in the search space will be overwhelmed by a drastic increase in search time caused by the sheer size of the formula that must be processed by the SAT solver. The only option left in such situations is to abandon the goal of full symmetry breaking and to re-phrase the objective of the exercise as that of minimizing SAT search time by breaking some, but not necessarily all, symmetries. Like a full SBP, a *partial* SBP selects the least assignment in each orbit of the symmetry group but may include other assignments as well. Formally, if $\hat{G}_\varphi$ is a subset of G_φ then

$$\rho\left(G_\varphi; X\right) \leqslant \rho\left(\hat{G}_\varphi; X\right) \tag{10.33}$$

A natural choice for $\hat{G}_\varphi$ is any set of irredundant generators for the group. Such a set is computationally attractive because, by Theorem 10.3.3, it is guaranteed to be exponentially smaller than the order of the symmetry group. Additionally, the independence of the generators in an irredundant set (i.e., the fact that none of the generators can be expressed in terms of any others) suggests that their corresponding SBP may block more symmetric assignments than that, say, of an equally-sized set of randomly-chosen elements of G_φ.

Table 10.11 lists the nine sets of irredundant generators for the symmetry group of the example function from Table 10.9 along with their corresponding SBPs. Three of these generator sets yield full SBPs, whereas the remaining nine produce partial SBPs. Interestingly, the set $\{\pi_4, \pi_5\}$ which does not generate the group also yields a full SBP! Thus, choosing to break the symmetries of a set of irredundant generators must be viewed as a heuristic whose effectiveness can only be judged empirically.

Table 10.11. Symmetry-breaking predicates of different generator sets for the symmetry group of the CNF formula in Table 10.9. Note that the SBPs of three generator sets achieves full symmetry breaking, whereas those of the remaining six achieve only partial symmetry breaking.

Generator Set	Lex-Leader Predicate	Symmetry Breaking
$\{\pi_1, \pi_2\}$	$\sum_{(a,b,c)} (0,1,2,3,6) = (a' + b)(a' + c')$	Partial
$\{\pi_1, \pi_3\}$	$\sum_{(a,b,c)} (0,1,2,6) = (a' + b)(b' + c')$	Full
$\{\pi_1, \pi_4\}$	$\sum_{(a,b,c)} (0,1,2,3,6) = (a' + b)(a' + c')$	Partial
$\{\pi_1, \pi_5\}$	$\sum_{(a,b,c)} (0,1,2,6) = (a' + b)(b' + c')$	Full
$\{\pi_2, \pi_3\}$	$\sum_{(a,b,c)} (0,1,2,4,6) = (a' + c')(b' + c')$	Partial
$\{\pi_2, \pi_4\}$	$\sum_{(a,b,c)} (0,1,2,3,6) = (a' + b)(a' + c')$	Partial
$\{\pi_2, \pi_5\}$	$\sum_{(a,b,c)} (0,1,2,4,6) = (a' + c')(b' + c')$	Partial
$\{\pi_3, \pi_4\}$	$\sum_{(a,b,c)} (0,1,2,6) = (a' + b)(b' + c')$	Full
$\{\pi_3, \pi_5\}$	$\sum_{(a,b,c)} (0,1,2,4,6) = (a' + c')(b' + c')$	Partial

10.7.3. Experimental Evaluation of Symmetry Breaking

The impact of symmetry breaking on SAT search time for the benchmark instances in Table 10.8 is shown in Table 10.12. For each instance, the table lists the following data:

- Instance parameters: family, name, number of variables, number of clauses, and number of literals
- SAT search time without symmetry breaking
- SBP size for each variant (mixed, variable, and value): number of variables, number of clauses, and number of literals
- SAT search time when the instance is conjoined with each of the three SBP variants

The SBPs in these experiments were created using the optimization described in [ASM03] which yields a permutation predicate consisting of $4|I|$ clauses and $14|I|$ literals where I is the index of irredundant bits specified in (10.26). SAT checking was carried out using the RSAT solver [RSA] which won the gold medal in the industrial category in the SAT'07 competition [SAT]. The experiments were performed on a 3GHz Intel Xeon workstation with 4GB RAM running the Redhat Linux Enterprise operating system. Each instance was allowed to run for up to 1000 seconds; times-outs are indicated in the table by '–'.

Close examination of these results leads to the following general observations:

- On average, the size of the SBP-augmented formula (number of variables and number of literals) is slightly more than twice the size of the original formula. The largest increase in size (about 3.4X for variables and 4X for literals) occurred for the Hole, FPGARoute, and ChnlRoute benchmarks; the smallest (about 1.3X for variables and 1.1X for literals) was for the Urq and Pipe benchmarks. These are relatively modest increases that should not tax the abilities of modern SAT solvers.
- Generally, but not always, the addition of the SBP helps to reduce the time to perform the SAT check. Specifically,
  - The Hole, ChnlRoute, Urq, and XOR instances all benefited from the addition of the SBP, even when the symmetry detection time was factored in. This was especially true for instances which could not be solved within the time-out limit.
  - The FPGARoute instances were quite easy to solve without the addition of the SBPs. There was a marginal speed-up in search time when the SBPs were added. However, when symmetry detection times were factored in, the net effect was a marked slow-down.
  - The results for the GRoute instances were mixed. The addition of SBPs improved search times in some cases but worsened it in others. When symmetry detection times were added in, however, the net effect was a noticeable slow-down.
  - The Pipe instances all suffered increases in search times when SBPs were added, even without considering symmetry detection times.
- In cases where only pure variable symmetries are present (Hole, ChnlRoute, FPGARoute, and GRoute), comparable search times were obtained for the two variants of an augmented instance (namely the instance augmented with the variable symmetry SBP and the one augmented with the mixed symmetry SBP).
- However, in cases where only pure value symmetries are present (Urq and all but the smallest XOR), the search times for instances augmented with mixed symmetry SBPs were worse (some timing out) than the search times for the same instances augmented with value symmetry SBPs. The only plausible explanation of such divergent behaviors is that the mixed and value symmetry SBPs are derived from two different sets of generators. This turned out to be the case as illustrated in Fig. 10.10. The figure shows the generators returned by **saucy** for the 46-variable Urq3_5 benchmark: part (a) shows the generators obtained for mixed symmetries and part (b) gives the generators obtained for value symmetries. In both cases, the generators consist of phase-shift permutations (as expected), but the two sets are different. This leads to different SBPs which, in this case, are simply conjunctions of negative literals that correspond to the phase-shift cycle of the variable that occurs first in the assumed variable ordering. Clearly, the generators obtained for value symmetries lead to a SBP that is superior to that obtained from the generators for mixed symmetries. Specifically, the SBP for value symmetries "fixes" 29 out of the 46 variables compared to the SBP for mixed symmetries which fixes only 12 variables. This difference in pruning power was not sufficient to cause the search times to be substantially different for this particular instance. For the larger instances, however, the pruning ability of the SBPs derived from value symmetries is clearly evident; in fact the SBPs derived from mixed symmetries were unsuccessful in helping the SAT solver prove the unsatisfiability of their corresponding instances within the 1000 second time-out.

10.8. Summary and a Look Forward

This chapter was concerned with recognizing symmetry in CNF formulas and exploiting it in the context of satisfiability checking. The particular approach we described for utilizing symmetry in SAT solving is sometimes referred as *static symmetry breaking* because it modifies the input to the SAT solver rather than modifying the SAT solver itself. Given a CNF formula $\varphi(X)$, static symmetry breaking consists of the following steps:

1. Converting the formula to a colored graph whose symmetries are isomorphic to the symmetries of the formula.
2. Computing the automorphism group G_φ of the graph and returning a set of generators $\hat{G}_\varphi \subset G_\varphi$ for it using a suitable graph automorphism program.
3. Mapping the returned generators back to symmetries of the formula.
4. Constructing an appropriate symmetry-breaking predicate $\rho(\hat{G}_\varphi; X)$ in CNF and conjoining it to the formula.
5. Checking $\varphi(X) \wedge \rho(\hat{G}_\varphi; X)$ using a suitable SAT solver.

The attractiveness of this approach stems from the fact that it can be used with *any* SAT solver and can, thus, automatically benefit from improvements in SAT solving technology. On the other hand, the method has a number of drawbacks that can limit its effectiveness in some cases. The method's inherent disadvantage is that the SBP created in step 4 relates to the *global symmetries* of the original formula. During SAT search, particular partial assignments to the variables may

Table 10.12. SAT search times (in seconds) with and without symmetry breaking.

Benchmark						Symmetry Breaking Predicate											
		Parameters			Time	Mixed Symmetries				Variable Symmetries				Value Symmetries			
Family	Name	Vars	Clauses	Lits		Vars	Clauses	Lits	Time	Vars	Clauses	Lits	Time	Vars	Clauses	Lits	Time
Hole	hole7	56	204	448	0.03	97	388	1358	0	97	388	1358	0	0	0	0	0.03
	hole8	72	297	648	0.18	127	508	1778	0	127	508	1778	0	0	0	0	0.18
	hole9	90	415	900	50.19	161	644	2254	0.01	161	644	2254	0	0	0	0	50.19
	hole10	110	561	1210	336.86	199	796	2786	0	199	796	2786	0.01	0	0	0	336.86
	hole11	132	738	1584	–	241	964	3374	0	241	964	3374	0	0	0	0	–
	hole12	156	949	2028	–	287	1148	4018	0	287	1148	4018	0	0	0	0	–
ChnlRoute	chnl10_11	220	1122	2420	–	508	2032	7112	0	508	2032	7112	0.01	0	0	0	–
	chnl10_12	240	1344	2880	–	556	2224	7784	0	556	2224	7784	0.01	0	0	0	–
	chnl10_13	260	1586	3380	–	604	2416	8456	0	604	2416	8456	0.01	0	0	0	–
	chnl11_12	264	1476	3168	–	614	2456	8596	0	614	2456	8596	0	0	0	0	–
	chnl11_13	286	1742	3718	–	667	2668	9338	0.01	667	2668	9338	0	0	0	0	–
	chnl11_20	440	4220	8800	–	1038	4152	14532	0	1038	4152	14532	0.01	0	0	0	–
FPGARoute	fpga10_8	120	448	1200	0.02	256	1024	3584	0	256	1024	3584	0	0	0	0	0.02
	fpga10_9	135	549	1485	0.01	223	892	3122	0	223	892	3122	0	0	0	0	0.01
	fpga12_8	144	560	1488	0.01	312	1248	4368	0	312	1248	4368	0	0	0	0	0.01
	fpga12_9	162	684	1836	0.04	273	1092	3822	0	273	1092	3822	0.01	0	0	0	0.04
	fpga12_11	198	968	2640	0.02	339	1356	4746	0.01	339	1356	4746	0	0	0	0	0.02
	fpga12_12	216	1128	3096	0.01	480	1920	6720	0.01	480	1920	6720	0	0	0	0	0.01
	fpga13_9	176	759	2031	0.01	299	1196	4186	0.01	299	1196	4186	0.01	0	0	0	0.01
	fpga13_10	195	905	2440	0.04	334	1336	4676	0.01	334	1336	4676	0	0	0	0	0.04
	fpga13_12	234	1242	3396	0.01	406	1624	5684	0	406	1624	5684	0.01	0	0	0	0.01
Groute	s3-3-3-1	864	7592	16038	0.08	528	2112	7392	0.74	528	2112	7392	0.12	0	0	0	0.08
	s3-3-3-3	960	9156	19258	0.06	720	2880	10080	0.16	720	2880	10080	0.16	0	0	0	0.06
	s3-3-3-4	912	8356	17612	0.82	600	2400	8400	0.34	600	2400	8400	0.09	0	0	0	0.82
	s3-3-3-8	912	8356	17612	0.41	648	2592	9072	0.27	648	2592	9072	1.25	0	0	0	0.41
	s3-3-3-10	1056	10862	22742	2.36	720	2880	10080	0.09	720	2880	10080	0.08	0	0	0	2.36
Urq	Urq3_5	46	470	2912	354.34	29	116	406	0.1	0	0	0	354.34	29	116	406	0
	Urq4_5	74	694	4272	–	43	172	602	122.73	0	0	0	–	43	172	602	0
	Urq5_5	121	1210	7548	–	72	288	1008	–	0	0	0	–	72	288	1008	0
	Urq6_5	180	1756	10776	–	109	436	1526	–	0	0	0	–	109	436	1526	0
	Urq7_5	240	2194	13196	–	143	572	2002	–	0	0	0	–	143	572	2002	0.01
	Urq8_5	327	3252	20004	–	200	800	2800	–	0	0	0	–	200	800	2800	0
XOR	x1_16	46	122	364	0.02	18	72	252	0	1	4	14	0.02	16	64	224	0.01
	x1_24	70	186	556	34.5	24	96	336	0.01	0	0	0	34.5	24	96	336	0
	x1_32	94	250	748	1.33	32	128	448	0	0	0	0	1.33	32	128	448	0
	x1_36	106	282	844	181.51	36	144	504	0.29	0	0	0	181.51	36	144	504	0
Pipe	2pipe_1_ooo	834	7026	19768	0.11	321	1284	4494	0.25	317	1268	4438	0.14	0	0	0	0.11
	2pipe_2_ooo	925	8213	23161	0.12	362	1448	5068	0.14	354	1416	4956	0.19	0	0	0	0.12
	2pipe	861	6695	18637	0.13	358	1432	5012	0.16	350	1400	4900	0.21	0	0	0	0.13


```
(1,-1)(5,-5)(12,-12)(13,-13)(19,-19)
(13,-13)(22,-22)(31,-31)(39,-39)(45,-45)
(2,-2)(5,-5)(13,-13)(19,-19)(22,-22)
(5,-5)(9,-9)(13,-13)(22,-22)(28,-28)(31,-31)(39,-39)(41,-41)
(1,-1)(11,-11)(22,-22)(31,-31)(32,-32)(39,-39)(41,-41)(44,-44)
(11,-11)(25,-25)(35,-35)
(10,-10)(11,-11)(13,-13)(22,-22)(29,-29)(31,-31)(39,-39)(41,-41)
(1,-1)(11,-11)(22,-22)(30,-30)(31,-31)(38,-38)(39,-39)(41,-41)
(16,-16)(22,-22)(23,-23)(37,-37)
(16,-16)(20,-20)(26,-26)(31,-31)(39,-39)(41,-41)
(16,-16)(17,-17)(25,-25)(31,-31)(39,-39)(41,-41)
(1,-1)(6,-6)(16,-16)(22,-22)(36,-38)
(5,-5)(13,-13)(23,-23)(28,-28)(42,-42)
(10,-10)(13,-13)(23,-23)(34,-34)
(1,-1)(21,-21)(23,-23)(38,-38)
(4,-4)(26,-26)(41,-41)
(3,-3)(25,-25)(41,-41)
(5,-5)(13,-13)(22,-22)(26,-26)(28,-28)(31,-31)(39,-39)(40,-40)(41,-41)
(8,-8)(25,-25)(26,-26)
(1,-1)(22,-22)(26,-26)(31,-31)(38,-38)(39,-39)(41,-41)(46,-46)
(1,-1)(5,-5)(13,-13)(28,-28)(36,-36)(44,-44)
(1,-1)(10,-10)(13,-13)(18,-18)(44,-44)
(15,-15)(38,-38)(44,-44)
(1,-1)(5,-5)(13,-13)(19,-19)(33,-33)(44,-44)
(14,-14)(19,-19)(28,-28)
(10,-10)(13,-13)(22,-22)(25,-25)(31,-31)(39,-39)(41,-41)(43,-43)
(5,-5)(7,-7)(13,-13)(19,-19)(22,-22)(25,-25)(31,-31)(39,-39)(41,-41)
(1,-1)(10,-10)(13,-13)(24,-24)(38,-38)
(5,-5)(10,-10)(19,-19)(27,-27)
```

(a) Generators for Mixed Symmetries with corresponding SBP $\bigwedge_{i \in S} x'_i$ where

$S = \{1\text{-}5, 8, 10, 11, 13\text{-}16\}$

```
(36,-36)(38,-38)(40,-40)(44,-44)(46,-46)
(32,-32)(34,-34)(35,-35)(38,-38)(40,-40)(42,-42)(43,-43)(44,-44)(46,-46)
(30,-30)(34,-34)(35,-35)(40,-40)(42,-42)(43,-43)(46,-46)
(29,-29)(35,-35)(43,-43)
(27,-27)(33,-33)(34,-34)(38,-38)(40,-40)(42,-42)(44,-44)(46,-46)
(25,-25)(26,-26)(34,-34)(40,-40)(42,-42)(43,-43)
(24,-24)(34,-34)(40,-40)(42,-42)(46,-46)
(22,-22)(23,-23)(26,-26)(31,-31)(39,-39)(40,-40)(41,-41)(42,-41)
(21,-21)(40,-40)(42,-42)(46,-46)
(20,-20)(37,-37)(40,-40)(42,-41)
(19,-19)(28,-28)(33,-33)(38,-38)(40,-40)(44,-44)(46,-46)
(18,-18)(34,-34)(38,-38)(40,-40)(42,-42)(44,-44)(46,-46)
(17,-17)(34,-34)(37,-37)(43,-43)
(16,-16)(26,-26)(31,-31)(37,-37)(39,-39)(40,-40)(41,-41)(42,-42)
(15,-15)(38,-38)(44,-44)
(14,-14)(33,-33)(38,-38)(40,-40)(44,-44)(46,-46)
(13,-13)(23,-23)(26,-26)(40,-40)(41,-41)(42,-42)(45,-45)
(12,-12)(33,-33)(44,-44)
(11,-11)(26,-26)(34,-34)(35,-35)(40,-40)(42,-42)(43,-43)
(10,-10)(26,-26)(34,-34)(40,-40)(41,-41)(42,-42)(45,-45)
(9,-9)(26,-26)(40,-40)
(8,-8)(34,-34)(40,-40)(42,-42)(43,-43)
(7,-7)(33,-33)(34,-34)(38,-38)(40,-40)(42,-42)(43,-43)(44,-44)(46,-46)
(6,-6)(37,-37)(40,-40)(42,-42)(46,-46)
(5,-5)(26,-26)(28,-28)(40,-40)(41,-41)(45,-45)
(4,-4)(26,-26)(41,-41)
(2,-2)(26,-26)(31,-31)(33,-33)(38,-38)(39,-39)(41,-41)(44,-44)(46,-46)
(3,-3)(26,-26)(34,-34)(40,-40)(41,-41)(42,-42)(43,-43)
(1,-1)(23,-23)(38,-38)(40,-40)(42,-42)(46,-46)
```

(b) Generators for Value Symmetries with corresponding SBP $\bigwedge_{i \in S} x'_i$ where

$S = \{1\text{-}22, 24, 25, 27, 29, 30, 32, 36\}$

Figure 10.10. Generators for the Urq3_5 benchmark

give rise to additional *local symmetries*, also known as *conditional symmetries* [BS07], that offer the possibility of further pruning of the search space. A preprocessing approach cannot utilize such symmetries. Computationally, even after applying the optimizations described in Sec. 10.7.1, the size of the SBP (expressed as a CNF formula) might still be too large to be effectively handled by a SAT solver. Furthermore, the pruning ability of the SBP depends on the generator set used to derive it and, as we have seen experimentally, can vary widely. Finally, the assumed, and essentially arbitrary, ordering of the variables to create the SBP may be at odds with the SAT solver's branching strategy; this limitation, however, is not as severe as the other two since modern SAT solvers employ adaptive decision heuristics that are insensitive to the initial static ordering of the variables. It may be useful, however, to explore orderings on the truth assignments that are not lexicographic. For example, orderings in which successive assignments have a Hamming distance of 1 (i.e., differ in only one bit position) might yield better SBPs because of better clustering of "least" assignments. Despite these shortcomings, static symmetry breaking is currently the most effective approach for exploiting symmetry in SAT search.

An alternative approach is to break symmetries *dynamically* during the search. In fact, one of the earliest references to utilizing symmetry in SAT advocated such an approach [BFP88]. However, except for a brief follow-up to this work in [LJPJ02], the early contributions of Benhamou et al in [BS92, BS94], and the more recent contribution of Sabharwal in [Sab05], there is practically little mention in the open literature of dynamic symmetry breaking in SAT. The situation is completely different in the domain of constraint satisfaction problems (CSP) where breaking symmetries during search has been a lively research topic for the past few years. One conjecture that might explain this anomaly is that modern implementations of conflict-driven backtrack SAT solvers are based on highly-optimized and finely-tuned search engines whose performance is adversely affected by the introduction of expensive symmetry-related computations in their innermost kernel.

The potential benefits of dynamic symmetry breaking are too compelling, however, to completely give up on such an approach. These benefits include the ability to identify local symmetries, and the possibility of boosting the performance of conflict-driven learning by recording the symmetric equivalents of conflict-induced clauses. Dynamic symmetry breaking is also appealing because it avoids the creation, as in static symmetry breaking, of potentially wasteful symmetry breaking clauses that drastically increase the size of the CNF formula while only helping marginally (or not at all) in pruning the search space. To realize these benefits, however, the integration of symmetry detection and symmetry breaking must be considered carefully. An intriguing possibility is to *merge* the graph automorphism and SAT search engines so that they can work simultaneously on the CNF formula as the search progresses. The recent improvements in graph automorphism algorithms, tailored specifically to the sparse graphs that characterize CNF formulas, suggest that this may now be a practical possibility. Further advances, including incremental updating of the symmetry group (through its generators) in response to variable assignments, are necessary, however, to make such a scheme effective on large CNF instances with many local

symmetries. In short, this is an area that is ripe for original research both in the graph automorphism space as well as in the SAT search space.

10.9. Bibliographic Notes

10.9.1. Theoretical Considerations

The study of symmetry in Boolean functions has a long history. Early work was primarily concerned with counting the number of symmetry types of Boolean functions of n variables. Assuming invariance under variable permutation and/or complementation, the 2^{2^n} possible functions of n variables can be partitioned into distinct equivalence classes. Slepian [Sle54] gave a formula for the number of such classes as a function of n. Slepian also related this to earlier work by Pólya [Pól40] and Young [You30], and noted that the group of permutations and/or complementations is isomorphic to the hyperoctahedral group (the automorphism group of the hyperoctahedron in n-dimensional Euclidean space) as well as to the group of symmetries of the n-dimensional hypercube. These investigations were further elaborated by Harrison [Har63] who gave formulas for separately counting the number of equivalence classes of Boolean functions under three groups: complementation, permutation, and combined complementation and permutation. He also pointed out that the combined group is the semi-direct, rather than the direct, product[8] of the other two. More recently, Chen [Che93] studied the *cycle structure* of the hyperoctahedral group by representing it using *signed permutations*, i.e., permutations on $\{1, 2, \ldots, n\}$ "with a + or − sign attached to each element $1, 2, \ldots, n$."

10.9.2. Symmetry in Logic Design

Symmetry of Boolean functions was also studied in the context of logic synthesis. Motivated by the desire to create efficient realizations of relay circuits for telephone switching exchanges, Shannon [Sha38, Sha49] developed a notation to describe and logically manipulate symmetric functions based on their so-called "a-numbers." Specifically, Shannon showed that a function which is symmetric in all of its variables (i.e., a function which remains invariant under all permutations of its variables) can be specified by a subset of integers from the set $\{0, 1, \ldots, n\}$. Such functions were termed *totally symmetric* in contrast with *partially symmetric* functions that remain invariant under permutation of only a *subset* of their variables. For example, $f(a, b, c) = ab + ac + bc$ is a totally symmetric function specified as $S_{\{2,3\}}(a, b, c)$ and interpreted to mean that f is equal to 1 when exactly 2 or exactly 3 of its variables are 1. The notation extends naturally when some variables need to be complemented before being permuted. Thus $f(a, b, c) = ab'c + a'bc + a'b'c'$ is specified as $S_{\{0,1\}}(a, b, c')$ meaning that f is

[8] A discussion of direct and semi-direct group products will take us too far afield. For current purposes, it is sufficient to note that a direct product is a special type of a semi-direct product. Specifically, the direct product of two groups G and H is the group whose underlying set is $G \times H$ and whose group operation is defined by $(g_1, h_1)(g_2, h_2) = (g_1 g_2, h_1 h_2)$ where $g_1, g_2 \in G$ and $h_1, h_2 \in H$.

equal to 1 when exactly 0 or exactly 1 of the literals from the set $\{a, b, c'\}$ is equal to 1. The specification of partially symmetric functions is more involved. Let $f(X, Y)$ be such a function where $\{X, Y\}$ is a partition of its variables such that the function remains invariant under all permutations of the X variables. Using the elementary symmetric functions[9] $S_{\{i\}}(X)$ for $i = 0, \ldots, |X|$ as an orthonormal basis, f can be specified as:

$$f(X, Y) = \sum_{0 \leqslant i \leqslant |X|} S_{\{i\}}(X) R_i(Y)$$

where $R_i(Y)$ is the *residual* function corresponding to $S_{\{i\}}(X)$.

Shannon's definition of symmetry yields a partition of a function's literals such that literals in a given cell of the partition can be permuted arbitrarily without causing the function to change. A variety of schemes were subsequently developed to derive this partition. All of these schemes are based on finding pairs of equivalent literals using the following "Boole/Shannon" expansion:

$$f(\ldots, x, \ldots, y, \ldots) = x'y' f_{x'y'} + x'y f_{x'y} + xy' f_{xy'} + xy f_{xy}$$

where $f_{x'y'}, \ldots, f_{xy}$ are the *cofactors* of f with respect to the indicated literals [BCH+82]. For example, $f_{x'y} = f(\ldots, 0, \ldots, 1, \ldots)$, i.e., f is restricted by the assignment of 0 to x and 1 to y. It is now straightforward to see that f remains invariant under a swap of x and y if and only if $f_{x'y} = f_{xy'}$. Similarly, f will be invariant under a swap of x and y' (resp. x' and y) if and only if $f_{x'y'} = f_{xy}$. In cycle notation,

$$(f_{x'y} = f_{xy'}) \Leftrightarrow (x, y)$$
$$(f_{x'y'} = f_{xy}) \Leftrightarrow (x, y')$$

Larger sets of symmetric literals can now be built from symmetric pairs using transitivity. The computational core in all of these schemes is the check for equivalence between the cofactor functions. Early work [Muk63] employed *decomposition charts* which, essentially, are truth tables in which the four combinations of the variable pair being checked for equivalence are listed row-wise whereas the combinations of the remaining variables are listed column-wise. Equality of cofactors is now detected as identical rows in the charts. Such a scheme, obviously, is not scalable and subsequent symmetry detection algorithms mirrored the evolution, over time, of more efficient representations for Boolean functions as cube lists (that encode functions in SOP form) [DS67], in terms of various spectral transforms [EH78, RF98], using Reed-Muller forms [TMS94], and finally as binary decision diagrams (BDDs) [MMW93, PSP94]. Most recently, the problem of detecting the symmetries of large Boolean functions was addressed by employing a portfolio of representations and algorithms including circuit data structures, BDDs, simulation, and (interestingly) satisfiability checking [ZMBCJ06].

It is important to note that the symmetries we have been discussing in this brief historical review are *semantic* rather than syntactic, i.e., symmetries that are independent of a function's particular representation. Detection of these symmetries, thus, requires some sort of *functional* analysis that checks various cofactors

[9] $S_A(X)$ is an elementary symmetric function if $|A| = 1$, i.e., the set of a-numbers A is a singleton.

for logical equivalence.[10] Furthermore, we may view the resulting partition of a function's literals into subsets of symmetric literals as an implicit compact representation of the function's symmetries. This is in contrast with the use of a set of irredundant generators to represent a group of symmetries. We should note, however, that a partition of the function's literals may not capture all possible symmetries of the function. This fact was "discovered" in the context of checking the logical equivalence of two functions with unknown correspondence between their inputs [MMM95], and led to augmenting Shannon's *classical symmetries*, which are based on a flat partitioning of a function's literals, with additional *hierarchical symmetries* that involve simultaneous swaps of sets of literals. These symmetries were further generalized in [KS00] where classical *first-order* symmetries (based on variable swaps) formed the foundation for *higher-order* symmetries that involve the swaps of both ordered and un-ordered sets of literals. These two extensions to Shannon's original definition of functional symmetry allowed the detection of many more function-preserving literal permutations in benchmark circuits. Still, such swap-based representations are inherently incapable of capturing arbitrary permutations such as $(a, b, c)(d, f, e)(g, h, m)(i, j, k)$ which is a symmetry of the function $ab + ac + af + aj + bc + be + bk + cd + ci + dk + ej + fi + gi + hj + km$ [Kra01].

10.9.3. Symmetry in Boolean Satisfiability Problems

Whereas interest in semantic symmetries, as noted above, was primarily motivated by the desire to optimize the design of logic circuits or to speed up their verification, interest in syntactic symmetries, i.e., symmetries of CNF formulas, arose in the context of constraint solving, specifically Boolean satisfiability checking. The earliest reference to the potential benefits of invoking symmetry arguments in logical reasoning is traditionally attributed to Krishnamurthy [Kri85] who introduced the notions of global and local "symmetry rules" that can be used to significantly shorten (from exponential to polynomial) the proofs of certain propositions such as the pigeon-hole principle. Another early reference to the use of symmetry in backtrack search is [BFP88] where symmetries are used *dynamically* by the search algorithm to restrict its search to the equivalence classes induced by the formula's symmetries (which are assumed to be given). In the same vain, Benhamou and Sais [BS92, BS94] described a modification to backtrack search that prunes away symmetric branches in the decision tree. Their method for detecting symmetries worked directly on the CNF formula and derived formula-preserving variable permutations incrementally through pair-wise substitutions. In [Sab05], Sabharwal described an interesting modification to the zChaff [MMZ+01] SAT solver that allows it to branch on a set of symmetric variables rather than on single variables. For instance, given a set $\{x_1, x_2, \ldots, x_k\}$ of k symmetric variables, a $k+1$-way branch sets $x_1, \ldots, x_i$ to 0 and $x_{i+1}, \ldots, x_k$ to 1 for each $i \in [0, k]$. This reduces the number of partial assignments that must

[10] As mentioned in Sec. 10.4, semantic symmetries can also be found by structural analysis of certain CNF representations of a function such as those that include a function's maxterms or its prime implicates. However, since these representations tend to be exponential in the number of variables, they are rarely used to find a function's semantic symmetries.

potentially be explored from 2^k to $k+1$. The identification of symmetric variables is assumed to be available from high-level descriptions of a problem (e.g., variables denoting pigeons in a pigeon-hole are all symmetric) and provided to the solver as an additional input.

The first complete exposition of the use of symmetries in a pre-processing step to prune the search space of SAT solvers must, however, be credited to Crawford et al. [Cra92, CGLR96]. In [Cra92], Crawford laid the theoretical foundation for reasoning by symmetry by showing that symmetry detection can be polynomially reduced to the problem of colored graph automorphism. The subsequent paper [CGLR96] detailed the basic flow of symmetry detection and breaking and introduced the following computational steps which, with some modification, still constitute the elements of the most successful approach, at least empirically, for the application of symmetry reasoning in SAT. Given a CNF formula:

1. Convert the formula to a colored graph whose automorphisms correspond to the symmetries of the formula.
2. Identify a set generators of the graph automorphism group using the **nauty** graph automorphism package [McK, McK81].
3. Using the generators, create a *lex-leader*[11] symmetry-breaking predicate from one of the following alternative sets of symmetries:
   - just the generators.
   - a subset of symmetries obtained from a truncated *symmetry tree* constructed from the generators.
   - a set of random group elements obtained from the generators.

The symmetry tree data structure in this flow was proposed as a mechanism to reduce the observed duplication in the SBP for problems with a large number of symmetries. However, except for some special cases (e.g., the full symmetry group of order $n!$ has a pruned symmetry tree with n^2 nodes), the size of the symmetry tree remains exponential in the number of variables.

The symmetry detection and breaking approach we described in Sec. 10.6 and Sec, 10.7 is based on [ARMS02, AMS03, ASM03] which extended Crawford's methodology in the following important ways:

- Crawford's graph construction in [CGLR96] used three node colors (for clauses, positive literals, and negative literals, respectively) and was, thus, limited to discovering variable symmetries. By using one color for both positive and negative literals, it was shown in [ARMS02] that additional symmetries can be found, namely value symmetries and their composition with variable symmetries. The restriction to just value symmetries by the use of a distinct color for each literal pair, as described in Sec. 10.6, is a further extension of these ideas that shows the versatility of the colored graph construction as a mechanism for studying the symmetries in a CNF formula.

[11] The term lex-leader indicates a lexicographic ordering of the truth assignments that is induced by an assumed ordering of the variables; it is the same as the numeric ordering in (10.5) corresponding to the interpretation of the truth assignments as unsigned integers.

- The size of Crawford's lex-leader SBP for breaking a single symmetry is quadratic in the number of variables. The optimizations introduced in [ARMS02, AMS03], and particularly in [ASM03], reduced this dependence to linear in a much smaller number that reflects the cycle structure of a symmetry as well as the presence of phase shifts in it. The relaxed permutation predicate in (10.32) represents a further optimization that shrinks the size of the SBP from $14|I|$ to $9|I|$ where I is the index set of irredundant bits defined by (10.26).
- The use of edges to represent binary clauses was suggested by Crawford to improve the efficiency of graph automorphism. The possibility that this optimization to the colored graph construction may produce spurious symmetries was noted in [ARMS03] and attributed to the presence of circular chains of implications in the CNF formula. Alternative constructions were also offered in [ARMS03] to deal with this problem, as well as suggestions for how to filter out the spurious symmetries when then arise.
- Finally, the work in [ARMS02, AMS03, ASM03] provided the first empirical large-scale validation of the feasibility of static symmetry breaking. This approach was validated empirically on a large number of different benchmark families.

Recent applications of this general symmetry breaking approach included its use in decision and optimization problems that involve pseudo-Boolean constraints (linear inequalities on Boolean variables) [ARSM07], and in deciding the satisfiability of quantified Boolean formulas [AJS07].

10.9.4. Symmetry in Constraint Satisfaction Problems

Research on the use of symmetry in constraint programming is too voluminous to adequately cover in a few paragraphs. Instead, we refer the reader to the excellent recent (and largely current) survey in [GPP06][12] and briefly highlight a few connections to similar issues in the context of Boolean satisfiability.

One of the questions that has preoccupied the constraint programming community over the past several years has been definitional: what exactly are the symmetries of a constraint programming problem? This question was largely settled in [CJJ+05] which provided a comprehensive taxonomy that encompassed most of the previous notions of CSP symmetry. In a nutshell, the type of symmetry in a CSP is classified based on:

- What is being permuted:
  - variables (yielding *variable* symmetries)
  - values (yielding *value* symmetries)
  - variable/value pairs
- What is invariant:
  - set of solutions (yielding *solution* symmetries)
  - set of constraints (yielding *constraint* symmetries)

[12] This survey also has pointers to key references on the use of symmetry in integer programming, planning, theorem proving, and model checking.

This classification is essentially the same as the one we used to categorize the symmetries of CNF formulas. Specifically, a CNF's semantic and syntactic symmetries correspond, respectively, to a CSP's solution and constraint symmetries. Furthermore, viewing a CSP's variable/value pair as a *literal*, we note that a CNF's variable, value, and mixed symmetries have their corresponding CSP counterparts. The similarity extends to the graph constructions used to extract the symmetries of a problem instance. For example, the constraint symmetries of a CSP instance correspond to automorphisms of the associated *microstructure* [JÓ3] which is the analog of the CNF formula graph we used in Sec. 10.6 to find the formula's syntactic symmetries.

There are some important differences, however. Symmetries tend to be easier to spot and describe in the CSP specifications of many problems than in the corresponding CNF encodings of these problems (the pigeon-hole problem being an obvious example.) There is room, thus, for hybrid approaches, along the lines of [Sab05], that combine high-level specifications of a problem's symmetries with a modern SAT solver's powerful search algorithms. Furthermore, many of the techniques for dynamic symmetry breaking that have been studied extensively in the CSP domain have yet to be fully explored in the CNF SAT context.

10.9.5. Graph Automorphism

Crawford's proposal in [CGLR96] for detecting and breaking the symmetries of CNF formulas included the use of McKay's graph isomorphism package, **nauty** [McK, McK81]. **nauty** had, by then, been the dominant publicly-available tool for computing automorphism groups, as well as for canonical labeling, of graphs. It was, thus, the natural choice for symmetry detection in subsequent extensions to Crawford's work in both the SAT and CSP domains. However, it quickly became clear that **nauty** was not optimized for the large sparse graphs constructed from CNF formulas that model design or verification problems. Specifically, the significant reductions in SAT search time that became possible because of SBP pruning were more than offset by **nauty**'s runtime for the detection of symmetries. This prompted the development of **saucy** [DLSM04] whose data structures were designed to take advantage of the sparsity in the graphs constructed from CNF formulas. **saucy** achieved significant speed-ups over **nauty** on such graphs and established the viability, at least for certain classes of problems, of the static symmetry detection and breaking approach. More enhancements along these lines were recently proposed in [JK07] which introduced the **bliss** tool for canonical labeling of large and sparse graphs. The latest offering in this space is a major enhancement to **saucy** that not only takes advantage of the sparsity of the input graph but also of the symmetry generators themselves [DSM08].

Acknowledgments

Many people helped in shaping the content of this chapter. I want to thank Igor Markov for nudging me to explore group theory as a way to describe and analyze symmetries of Boolean functions. Fadi Aloul was tremendously helpful in providing the experimental data for the mixed, variable, and value symmetries of CNF formulas. Paul Darga helped improve the description of the basic graph automorphism algorithm. Mark Liffiton provided the examples used to illustrate the various concepts. Zaher Andraus provided detailed comments from a careful reading of the manuscript. I would also like to thank my editor Toby Walsh for his insightful feedback. This work was completed during the author's sabbatical leave at Carnegie Mellon in Qatar and was supported, in part, by the National Science Foundation under ITR grant No. 0205288.

References

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[AJS07] G. Audemard, S. Jabbour, and L. Sais. Symmetry breaking in quantified boolean formulae. In *Proc. International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 2262–2267, 2007.

[AMS03] F. A. Aloul, I. L. Markov, and K. A. Sakallah. Shatter: Efficient symmetry-breaking for boolean satisfiability. In *Proc. 40th IEEE/ACM Design Automation Conference (DAC)*, pages 836–839, Anaheim, California, 2003.

[ARMS02] F. A. Aloul, A. Ramani, I. Markov, and K. A. Sakallah. Solving difficult sat instances in the presence of symmetry. In *Proc. 39th IEEE/ACM Design Automation Conference (DAC)*, pages 731–736, New Orleans, Louisiana, 2002.

[ARMS03] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. Solving difficult instances of boolean satisfiability in the presence of symmetry. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(9):1117–1137, 2003.

[ARSM07] F. A. Aloul, A. Ramani, K. A. Sakallah, and I. L. Markov. Symmetry breaking for pseudo-boolean constraints. *ACM Journal of Experimental Algorithms*, 12(Article No. 1.3):1–14, 2007.

[ASM03] F. A. Aloul, K. A. Sakallah, and I. L. Markov. Efficient symmetry breaking for boolean satisfiability. In *Proc. 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 271–282, Acapulco, Mexico, 2003.

[BCH+82] R. K. Brayton, J. D. Cohen, G. D. Hachtel, B. M. Trager, and D. Y. Y. Yun. Fast recursive boolean function manipulation. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 58–62, Rome, Italy, 1982.

[BFP88] C. A. Brown, L. Finkelstein, and P. W. Purdom. Backtrack searching in the presence of symmetry. In T. Mora, editor, *6th International Conference on Applied Algebra, Algebraic Algorithms and Error Correcting Codes*, pages 99–110, 1988.

[BS92] B. Benhamou and L. Saïs. Theoretical study of symmetries in propositional calculus and applications. *Lecture Notes In Computer Science*, 607:281–294, 1992. 11th International Conference on Automated Deduction (CADE-11).

[BS94] B. Benhamou and L. Saïs. Tractability through symmetries in propositional calculus. *Journal of Automated reasoning*, 12:89–102, 1994.

[BS07] B. Benhamou and M. R. Saidi. Local symmetry breaking during search in csps. *Lecture Notes in Computer Science*, LNCS 4741:195–209, 2007.

[Can01] E. R. Canfield. Meet and join within the lattice of set partitions. *The Electronic Journal of Combinatorics*, 8:1–8, #R15, 2001.

[CBB00] B. Cheswick, H. Burch, and S. Branigan. Mapping and visualizing the internet. In *USENIX Annual Technical Conference*, pages 1–13, 2000.

[CGLR96] J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Principles of Knowledge Representation and Reasoning (KR'96)*, pages 148–159, 1996.

[Che93] W. Y. C. Chen. Induced cycle structures of the hyperoctahedral group. *SIAM J. Disc. Math.*, 6(3):353–362, 1993.

[CJJ+05] D. Cohen, P. Jeavons, C. Jefferson, K. E. Petrie, and B. M. Smith. Symmetry definitions for constraint satisfaction problems. *Lecture Notes in Computer Science*, LNCS 3709:17–31, 2005.

[Cra92] J. Crawford. A theoretical analysis of reasoning by symmetry in first-order logic (extended abstract). In *AAAI-92 Workshop on Tractable Reasoning*, pages 17–22, San Jose, CA, 1992.

[DIM] DIMACS challenge benchmarks. `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/`.

[DLSM04] P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov. Exploiting structure in symmetry detection for cnf. In *Proc. 41st IEEE/ACM Design Automation Conference (DAC)*, pages 530–534, San Diego, California, 2004.

[DS67] D. L. Dietmeyer and P. R. Schneider. Identification of symmetry, redundancy and equivalence of boolean functions. *IEEE Trans. on Electronic Computers*, EC-16(6):804–817, 1967.

[DSM08] P. T. Darga, K. A. Sakallah, and I. L. Markov. Faster symmetry discovery using sparsity of symmetries. In *Proc. 45th IEEE/ACM Design Automation Conference (DAC)*, pages 149–154, Anaheim, California, 2008.

[EH78] C. R. Edwards and S. L. Hurst. A digital synthesis procedure under function symmetries and mapping methods. *IEEE Trans. on Computers*, C-27(11):985–997, 1978.

[Fra00] J. B. Fraleigh. *A First Course in Abstract Algebra.* Addison Wesley Longman, Reading, Massachusetts, 6th edition, 2000.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[GPP06] I. P. Gent, K. E. Petrie, and J.-F. Puget. Symmetry in constraint programming. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint programming.* Elsevier, 2006.

[GT00] R. Govindan and H. Tangmunarunkit. Heuristics for internet map discovery. In *IEEE INFOCOM*, pages 1371–1380, 2000.

[Har63] M. A. Harrison. The number of transitivity sets of boolean functions. *Journal of the Society for Industrial and Applied mathematics*, 11(3):806–828, 1963.

[ISP] ISPD 2005 placement competition. `http://www.sigda.org/ispd2005/contest.htm`.

[JÓ93] P. Jégou. Decomposition of domains based on the micro-structure of finite constraint-satisfaction problems. In *AAAI'93*, pages 731–736, 1993.

[JK07] T. Junttila and P. Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *Ninth Workshop on Algorithm Engineering and Experiments (ALENEX 07)*, New Orleans, LA, 2007.

[Kra01] V. N. Kravets. *Constructive Multi-Level Synthesis by Way of Functional Properties.* PhD thesis, University of Michigan, 2001.

[Kri85] B. Krishnamurthy. Short proofs for tricky formulas. *Acta Informatica*, 22:253–275, 1985.

[KS00] V. N. Kravets and K. A. Sakallah. Generalized symmetries in boolean functions. In *Digest of IEEE International Conference on Computer-Aided Design (ICCAD)*, pages 526–532, San Jose, California, 2000.

[LJPJ02] C. M. Li, B. Jurkowiak, and P. W. Purdom Jr. Integrating symmetry breaking into a dll procedure. In *International Symposium on Boolean Satisfiability (SAT)*, pages 149–155, Cincinnatti, OH, 2002.

[McK] B. D. McKay. nauty user's guide (version 2.2). `http://cs.anu.edu.au/~bdm/nauty/nug.pdf`.

[McK81] B. D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[MMM95] J. Mohnke, P. Molitor, and S. Malik. Limits of using signatures for permutation independent boolean comparison. In *Asia South Pacific Design Automation Conference*, pages 459–464, 1995.

[MMW93] D. Möller, J. Mohnke, and M. Weber. Detection of symmetry of boolean functions represented by robdds. In *International Conference on Computer Aided Design*, pages 680–684, 1993.

[MMZ+01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Design Automation Conference*, pages 530–535, Las Vegas, 2001.

[Muk63] A. Mukhopadhyay. Detection of total or partial symmetry of a switching function with the use of decomposition charts. *IEEE Trans. on Electronic Computers*, EC-12(5):553–557, 1963.

[NASR01] G.-J. Nam, F. Aloul, K. A. Sakallah, and R. Rutenbar. A comparative study of two boolean formulations of fpga detailed routing constraints. In *International Sympoisum on Physical Design (ISPD '01)*, pages 222–227, Sonoma, California, 2001.

[Pól40] G. Pólya. Sur les types des propositions composées. *J. Symbolic Logic*, 5:98–103, 1940.

[PSP94] S. Panda, F. Somenzi, and B. F. Plessier. Symmetry detection and dynamic variable ordering of decision diagrams. In *International Conference on Computer-Aided Design*, pages 628–631, 1994.

[Qui52] W. V. Quine. The problem of simplifying truth functions. *Amer. Math. Monthly*, 59(8):521–531, 1952.

[Qui55] W. V. Quine. A way o simplify truth functions. *Amer. Math. Monthly*, 62(9):627–631, 1955.

[Qui59] W. V. Quine. On cores and prime implicants of truth functions. *Amer. Math. Monthly*, 66(9):755–760, 1959.

[RF98] S. Rahardja and B. L. Falkowski. Symmetry conditions of boolean functions in complex hadamard transform. *Electronic Letters*, 34:1634–1635, 1998.

[RSA] RSAT. `http://reasoning.cs.ucla.edu/rsat/`.

[Sab05] A. Sabharwal. Symchaff: A structure-aware satisfiability solver. In *American Association for Artificial Intelligence (AAAI)*, pages 467–474, Pittsburgh, PA, 2005.

[SAT] SAT competition. `http://www.satcompetition.org`.

[Sha38] C. E. Shannon. A symbolic analysis of relay and switching circuits. *Trans. AIEE*, 57:713–723, 1938.

[Sha49] C. E. Shannon. The synthesis of two-terminal switching circuits. *Bell Systems Technical Journal*, 28(1):59–98, 1949.

[Sle54] D. Slepian. On the number of symmetry types of boolean functions of n variables. *Canadian J. Math.*, 5:185–193, 1954.

[TMS94] C.-C. Tsai and M. Marek-Sadowska. Detecting symmetric variables in boolean functions using generalized reed-muller forms. In *International Conferece on Circuits and Systems*, volume 1, pages 287–290, 1994.

[Urq87] A. Urquhart. Hard examples for resolution. *Journal of the Association for Computing Machinery*, 34(1):209–219, 1987.

[U.S] U.S. census bureau. `http://www.census.gov/geo/www/tiger/tigerua/ua_tgr2k.html`.

[VB01] M. N. Velev and R. E. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. In *Proc. Design Automation Conference (DAC)*, pages 226–231, New Orleans, Louisiana, 2001.

[You30] A. Young. On quantitative substitutional analysis. *Proc. London Math. Soc. (2)*, 31:273–288, 1930.

[ZMBCJ06] J. S. Zhang, A. Mishchenko, R. Brayton, and M. Chrzanowska-Jeske. Symmetry detection for large boolean functions using circuit representation, simulation, and satisfiability. In *Design Automation Conference*, pages 510–515, 2006.

Handbook of Satisfiability
Armin Biere, Marijn Heule, Hans van Maaren and Toby Walsh (Eds.)
IOS Press, 2009

doi:10.3233/978-1-58603-929-5-339

Chapter 11

Minimal Unsatisfiability and Autarkies

Hans Kleine Büning and Oliver Kullmann

The topic of this chapter is the study of certain forms of "redundancies" in propositional conjunctive normal forms and generalisations. In Sections 11.1 - 11.7 we study "minimally unsatisfiable conjunctive normal forms" (and generalisations), unsatisfiable formulas which are irredundant in a strong sense, and in Sections 11.8 - 11.13 we study "autarkies", which represent a general framework for considering redundancies. Finally in Section 11.14 we collect some main open problems.

11.1. Introduction

A literal is a variable or a negated variable. Let X be a set of variables, then $\mathrm{lit}(X)$ is the set of literals over the variables in X. Clauses are disjunctions of literals. Clauses are also considered as sets of literals. A propositional formula in conjunctive normal form (CNF) is a conjunction of clauses. CNF formulas will be considered as multi-sets of clauses. Thus, they may contain multiple occurrences of clauses. The set of all variables occurring in a formula φ is denoted as $\mathrm{var}(\varphi)$. Next we introduce formally the notion of *minimal unsatisfiability*. Please note that in some of the older papers minimal unsatisfiable formulas are sometimes called "critical satisfiable".

Definition 11.1.1. A set of clauses $\{f_1, \ldots, f_n\} \in \mathrm{CNF}$ is called *minimal unsatisfiable* if $\{f_1, \ldots, f_n\}$ is unsatisfiable and for every clause f_i $(1 \leq i \leq n)$ the formula $\{f_1, \ldots, f_{i-1}, f_{i+1}, \ldots, f_n\}$ is satisfiable. The set of minimal unsatisfiable formulas is denoted as MU.[1]

The complexity class D^P is the set of problems which can be described as the difference of two NP-problems, i.e., a problem Z is in D^P iff $Z = X - Y$ where X and Y are in NP. The D^P completeness of MU has been shown by a reduction of the D^P-complete problem UNSAT-SAT [PW88]. UNSAT-SAT is the set of pairs (F, G) for which F is unsatisfiable and G is satisfiable.

Theorem 11.1.1. *[PW88]* MU *is D^P-complete.*

[1] Depending on the author, instead of "minimal unsatisfiable formulas" also "minimally unsatisfiable formulas" can be used (or is preferable).