Computable Invariants for Quandles

A Senior Project submitted to
The Division of Science, Mathematics, and Computing
of
Bard College

by
Zhiwei Steven Wu

Annandale-on-Hudson, New York
May, 2012

Abstract

This senior project focuses on a collection of algebras that arise in knot theory called quandles. In particular, we study how to distinguish non-isomorphic quandles using different computational methods. Most of this work is built on The Library for Automated Deduction Research (LADR). Within LADR, the program `isofilter` removes the isomorphic duplicates for a collection of first-order models, but it works very slowly in general. In this senior project, we propose an augmented algorithm to filter quandles using computable invariants: we separate a stream of input quandles into different buckets based on the invariants; then we apply `isofilter` to the buckets containing more than one quandles. In fact, our invariant checking can distinguish most of the isomorphism classes for quandles up to size 30. It thus improves the quandle filtering process by largely reducing the use of `isofilter`.

Contents

List of Figures

Dedication

To my parents, for supporting me financially, mentally and unconditionally.

Also to William McCune, for his wonderful LADR.

Acknowledgments

First, I would like thank my two advisors James Belk and Robert McGrail for their guidance in my senior project, academic career and beyond. I also want to thank my academic advisor Gregory Landweber for the past four years' support and encouragement in my study of mathematics, physics and computer science. For all of the understanding, companionship and inspiration, I would like to thank my friends and mentors:

Anis
Becker
Chu
Dev
E. Fiorini
Feifan
Gu
H. Lin
J. Goh
Larson
Maria Belk
Pencil
S-Y. Lee
von Stengel
Xu $\times$ 2
Yen
Zana

1
Introduction

This senior project is motivated by previous work on quandles in The Laboratory for Algebraic and Symbolic Computation. We have been investigating the constraint satisfaction problem (CSP) over finite quandles. The relationship between the CSP and finite algebras allows us to assign to each finite algebra some subclass of NP. It was discovered that disconnected quandles are NP-complete, and hence classified [2]. Therefore, we can assume quandles are connected without loss of generality. In order to study connected quandles extensively, we need to be able to distinguish different isomorphism classes.

Within The Library for Automated Deduction Research (LADR), the program `isofilter` is a tool for removing duplicates (up to isomorphism) from a collection of first-order models, but it is very slow even for small collection of small-sized quandles. In order to assist `isofilter` to distinguish different quandle isomorphism classes, it is necessary to find invariants that can separate a stream of quandles into different buckets, where distinct buckets cannot have isomorphic quandles. We would gain a better run-time performance if we only apply `isofilter` to smaller buckets instead of the entire list of input quandles.

Connected quandles are well-behaved because we can associate each isomorphism class with a cycle structure. With this cycle structure as an invariant, we are able to distinguish some of the isomorphism classes. However, this invariant is not strong enough for our purposes, so we need to look for new invariants. Researchers have been studying various quandle invariants, and in this paper we introduce quandle identities as invariants. The idea of using such invariants is very simple: we generate a set of identities and test to see which identities each quandle satisfies. Since isomorphic quandles satisfy the same identities, we can separate two quandles in different buckets if they satisfy different subset of identities. This procedure can be nicely implemented with another program `clausetester` in LADR. With the aid of these invariants, we implemented an augmented quandle filtering algorithm by incorporating `isofilter`, `clausetester`, Prolog and GAP4. It turns out that we can distinguish most of the isomorphism classes for quandles up to size 30 just using our invariants.

This senior project is structured as follows. Chapter 2 gives a brief introduction to quandles along with important definitions and examples. In Chapter 3, we will formulate the notion of permutation signature, which is the cycle structure invariant for a quandle as mentioned. Permutation signature provides a useful tool to distinguish some of the isomorphism classes, but we will also show its limitations. At the end of this chapter, an algorithm for computing the permutation signature is included.

In Chapter 4, we will present a new kind of quandle invariants — normal form identities. We will first define normal form terms using normal form rewriting rules introduced by Quay-De La Vallee [7], and in turn define normal form identities. We would also talk about how to generate normal identities and remove equivalent identities using Prolog.

In Chapter 5, we present a new algorithm that integrates our invariants with `isofilter`. The implementation details of each computational tool would be included. In the last chapter, we will analyze the efficiency of this algorithm, and different factors that would affect the run-time performance. We also propose some potential improvements to our quandle filtering algorithm.

2
Quandles

The earliest work on **racks** is due to the unpublished correspondence during 1959 between John Conway and Gavin Wraith, who at the time were undergraduate students at the University of Cambridge. In a 1982 paper[5], David Joyce introduced a special case of rack called **quandle** in a context of knot theory. A quandle's axioms are representations of the three Reidemeister moves, which makes it a natural source of knot invariants. Over the past decades, the study of quandles as knot invariants has provided a very useful tool for knot theory.

2.1 Basic definitions

Definition 2.1.1. A **quandle** $(Q, *, /)$ is a set Q together with two binary operations: $*$ and $/ \colon Q \times Q \to Q$ satisfying the following axioms:

Idempotence: $\forall x (x * x = x)$

Right Cancellation: $\forall x\, y ((x * y)/y = x)$

Right Cancellation: $\forall x\, y ((x/y) * y = x)$

Right Self-Distributivity: $\forall x\, y\, z ((x * y) * z = (x * z) * (y * z))$. △

The simplest quandles we can form are the unary quandles.

Example 2.1.2. An **unary quandle** or a **trivial quandle** of size n, denoted by $\mathbf{U_n}$, is a quandle such that $a * b = a$ for all $a, b \in U_n$. The unary quandle U_3 has the following multiplication table for operation $*$.

$*$	0	1	2
0	0	0	0
1	1	1	1
2	2	2	2

$\Diamond$

Since the / operation can be determined by the $*$ operation according to Definition 2.1.1, we can describe a quandle only using the $*$ multiplication table.

Lemma 2.1.3. *A quandle Q also satisfies the following:*

1. $x/x = x$;
2. $(x/y) * z = (x * z)/(y * z)$;
3. $(x/y)/z = (x/z)/(y/z)$;
4. $(x * y)/z = (x/z) * (y/z)$.

Proof. Using the quandle axioms, we have

$$\begin{aligned} x/x &= ((x * x)/x) = x; \\ (x/y) * z &= (((x/y) * z) * (y * z))/(y * z) \\ &= (((x/y) * y) * z)/(y * z) = (x * z)/(y * z); \\ (x/y)/z &= (((x/z) * z)/((y/z) * z))/z \\ &= (((x/z)/(y/z)) * z)/z = (x/z)/(y/z); \\ (x * y)/z &= (((x * y)/z)/(y/z)) * (y/z) \\ &= (((x * y)/y)/z) * (y/z) = (x/z) * (y/z). \end{aligned}$$

We can see the right self-distributivity also works for any combination of operations. $\square$

Definition 2.1.4. Let H be a nonempty subset of a quandle Q. Then H is a **subquandle** of Q if, under the binary operations on Q, H itself forms a quandle. △

Definition 2.1.5. The **direct product** of quandles $Q_1, Q_2 \dots Q_n$ is a quandle defined as $Q_1 \times Q_2 \cdots \times Q_n = \{(q_1, q_2, \dots, q_n) \mid q_1 \in Q_1, q_2 \in Q_2, \dots, q_n \in Q_n\}$ with the following binary operations:

$$(q_1, q_2, \dots, q_n) * (q_1', q_2', \dots, q_n') = (q_1 * q_1', q_2 * q_2', \dots, q_n * q_n')$$
$$(q_1, q_2, \dots, q_n)/(q_1', q_2', \dots, q_n') = (q_1/q_1', q_2/q_2', \dots, q_n/q_n').$$

△

Quandle also has its own definition of "abelian".

Definition 2.1.6. A quandle Q is **medial** if $(a * b) * (c * d) = (a * c) * (b * d)$ for all $a, b, c, d \in Q$. △

Definition 2.1.7. A quandle Q is **involutory** if $(x * y) * y = x$ for all $x, y \in Q$. △

Definition 2.1.8. Let Q be a quandle. The **right Cayley graph** of Q is a graph with one vertex for each element of Q, and an edge from a to $a * b$ for all $a, b \in Q$. △

Definition 2.1.9. A **connected quandle** is a quandle with a connected right Cayley graph. △

This work is primarily concerned with connected quandles.

2.2 Quandle Examples

As shown in Section 2.1, some of the definitions in quandles are analogous to the ones for groups. We can also define a quandle for each group, and the two binary operations are associated with conjugations.

Example 2.2.1 (Conjugation Quandle)**.** Let G be a group, the **conjugation quandle** of G, denoted **Conj(G)**, is a quandle $(G, *, /)$ with the operations defined as $g * h = h^{-1}gh \ , g/h = hgh^{-1}$. Indeed, each $Conj(G)$ satisfies the quandle axioms in Definition 2.1.1:

- $x * x = x^{-1}\,x\,x = x$
- $(x * y)/y = y\,(y^{-1}\,x\,y)\,y^{-1} = x$
- $(x/y) * y = y^{-1}\,(y\,x\,y^{-1})\,y = x$
- $(x * y) * z = z^{-1}(y^{-1}\,x\,y)\,z = (z^{-1}\,y^{-1}\,z)\,(z^{-1}\,x\,z)\,(z^{-1}\,y\,z) = (x * z) * (y * z)$

$\Diamond$

Example 2.2.2 (Transposition Quandle)**.** Let $Sym(n, 2)$ denote the set of all transpositions in the symmetric group S_n. The **transposition quandle** of order $\frac{n(n-1)}{2}$, denoted by T_n, is defined as $Conj(Sym(n, 2))$. $\Diamond$

Example 2.2.3 (Forbidden Group Quandle)**.** Not all quandles are conjugation quandles. Joyce[5] observes that the quandle with the following multiplication table is not a conjugation quandle.

*	0	1	2
0	0	0	1
1	1	1	0
2	2	2	2

We call this quandle the **forbidden group quandle**. Joyce[5] also shows that a quandle Q can be written as $Conj(G)$ for some group G if and only if it does not contain the forbidden group quandle as a subquandle. $\Diamond$

Example 2.2.4 (Tait Quandle)**. Tait quandle** is a quandle of size 3 with its right Cayley Graph (ignore the loop connecting vertices to themselves) in Figure 2.2.1. And the multiplication table:

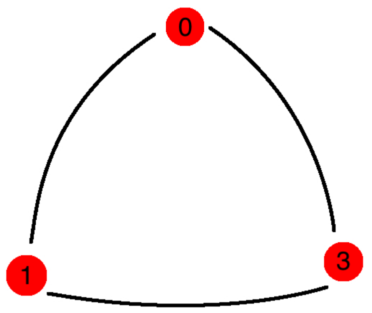

Figure 2.2.1. Right Cayley Graph for T_3

$*$	0	1	2
0	0	2	1
1	2	1	0
2	1	0	2

Tait quandle is used for tricoloring knots, and tricolorability is an invariant among knots. Notice that Tait quandle is denoted T_3 because it is also a transposition quandle.

Example 2.2.5 (Latin Quandle)**.** A quandle Q is **Latin** if $x * y = x * z$ implies $y = z$ for all elements in Q. This means the multiplication tables of Latin quandles are Latin squares because each row and column is a permutation of the quandle elements. In other words, a Latin quandle is a quandle that satisfies the following Latin condition.

Latin Condition $\forall x, y \in Q, \exists! z \in Q$ such that $x * z = y$. $\Diamond$

Corollary 2.2.6 (Latin Connectivity Theorem)**.** *Every Latin quandle is connected.*

Proof. Let $x \in Q$. Since Q is a latin quandle, every other elements in Q can be reached from x by operating once by some element in Q. Thus, the right Cayley graph of Q has only one connected component, and hence it is connected. $\square$

Example 2.2.7. A **dihedral quandle** of size n with elements $\{0, 1 \ldots, n-1\}$, denoted **Dih**(**n**), is a quandle that satisfies $i*j = 2j-i \bmod n$, where $-$ denotes the usual operation of modular arithmetic. Similar to the dihedral groups, dihedral quandles also reflect the symmetries of a regular polygon. The dihedral quandle of size 3 captures the reflections

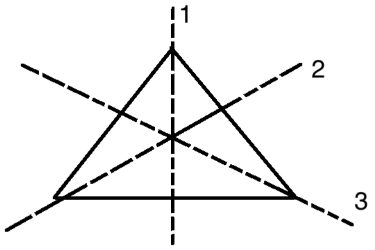

Figure 2.2.2. A Triangle and its Bisectors

among the three angle bisectors in the triangle. For example, in Figure 2.2.2 the reflection of line 1 about line 2 is line 3, and in the dihedral quandle we have $1 * 2 = 2 \times 2 - 1 \equiv 0$ mod 3. $\Diamond$

Example 2.2.8. A **linear quandle** $\mathbf{LQ(n, k)}$ is a quandle with elements $\{0, 1, \ldots, n-1\}$ that satisfies $x * y = k(x - y) + y \mod n$. For example, the quandle $LQ(5, 2)$ has the following multiplication table:

*	0	1	2	3	4
0	0	4	3	2	1
1	2	1	0	4	3
2	4	3	2	1	0
3	1	0	4	3	2
4	3	2	1	0	4

Both unary quandles and dihedral quandles are linear quandles. In particular

$$LQ(n, n-1) \cong Dih(n) \text{ and } LQ(n, 1) \cong U_n.$$

$\Diamond$

We can generalize the idea of linear quandles to the next type of quandles.

Example 2.2.9. Let $(A, +)$ be an abelian group and $t\colon A \to A$ be an automorphism map. An **Alexander quandle** $Q = (A, *, /)$ is a quandle with the following operations

- $x * y = t(x - y) + y$

- $x/y = t^{-1}(x-y)+y$.

In particular, a dihedral quandle is an Alexander quandle induced by a cyclic group. $\Diamond$

Theorem 2.2.10. *Let Q be a finite Alexander quandle corresponding to an automorphism t. Q is connected if and only if the map $1-t$ is also an automorphism.*

Proof. Suppose that $1-t$ is not an automorphism. Since Q is finite and $1-t$ is a map from Q to Q, we know that $1-t$ is neither one-to-one nor onto. It follows that the image of $1-t$ is a strict subset of Q. Let $a, b \in Q$, we can write

$$\begin{aligned}(1-t)a * b &= t\left((1-t)a - b\right) + b \\ &= t(1-t)a + (1-t)b \\ &= (1-t)(ta+b)\end{aligned}$$

Thus, the element $(1-t)a$ is only connected to elements in the image of $1-t$. Also, for an arbitrary element $c \in Q$, we have $b' = c - ta$ such that $(1-t)a * b' = (1-t)c$. Therefore, any two elements in the image of $1-t$ is connected, so the image forms a connected component in the right Cayley graph of Q. Since such a component is a strict subgraph by assumption, we know that the graph is not connected and Q is not connected.

Suppose $1-t$ is an automorphism. Then for any $x, z \in Q$ there exists a $y \in Q$ such that $(1-t)y = z - tx$ such that $x * y = t(x-y)+y = tx + (1-t)y = tx + z - tx = z$. Therefore, Q is a Latin quandle. By Corollary 2.2.6, we know that Q is connected. $\square$

2.3 Congruence Relations

Definition 2.3.1. Let $\equiv$ be an equivalence relation on Q. Then $\equiv$ is a **congruence relation** on Q if

$$x \equiv x', y \equiv y' \text{ implies } x * y \equiv x' * y'.$$

$\triangle$

Definition 2.3.2. Given a congruence relation $\equiv$ on a quandle Q, the quotient of Q modulo the congruence relation $\equiv$ is denoted by $Q/\!\equiv$ and is defined by

$$Q/\!\equiv \; = \{[x] \mid x \in Q\}.$$

where $[x]$ is the congruence class of x.

△

Example 2.3.3. There are different congruence relations defined on a quandle. The most obvious congruence relations are two trivial congruence relations on a quandle Q

- $a \equiv b$ for all $a, b \in Q$
- $a \equiv b$ if and only if $a = b$.

We also have **component congruence**: $a \equiv b$ if and only if a and b are in the same connected component. ◊

Corollary 2.3.4. *Let C denote the component congruence relation on a quandle Q of size n. Then we know that $Q/C \cong U_n$.*

Proof. We know that each connected congruence class of Q is a connected component. Thus, no two elements in Q/C is connected, so $Q/C \cong U_n$. □

Example 2.3.5. Let F be the forbidden group quandle in Example 2.2.3, and C denote the component congruence relation. From the multiplication table we mentioned above, we know it has congruence classes $A = \{0, 1\}$ and $B = \{2\}$. Then, F/C has the following multiplication table:

*	A	B
A	A	A
B	B	B

◊

3
Permutation Signature

In this chapter, we will introduce the notion of permutation signature for connected quandles. Permutation signature can serve as a useful invariant for distinguishing some of the quandle isomorphism classes.

3.1 Inner Automorphism

Definition 3.1.1. Let Q and Q' be quandles. A **quandle homomorphism** is a function $r_q\colon Q \to Q'$ such that $r_q(x * y) = r_q(x) * r_q(y)$ and $r_q(x/y) = r_q(x)/r_q(y)$ for all $x, y \in Q$.
A **quandle isomorphism** is a bijective quandle homomorphism.
A **quandle automorphism** is a quandle isomorphism from a quandle to itself. The group of all of Q's automorphisms is called the **automorphism group** of Q, denoted $Aut(Q)$. △

Theorem 3.1.2 (Inner Automorphism Theorem)**.** *For any $q \in Q$, the map $r_q\colon Q \to Q$ defined by $r_q(x) = x * q$ is an automorphism of the quandle Q.*

Proof. First, we can show that r_q is a homomorphism. By right self-distributive law, we know that

$$r_q(x * y) = (x * y) * q = (x * q) * (y * q) = r_q(x) * r_q(y)$$
$$r_q(x/y) = (x/y) * q = (x * q)/(y * q) = r_q(x)/r_q(y).$$

Since for each $y \in Q$, we have $x = y/q$ such that $r_q(x) = (y/q) * q = y$, we know that the map r_q is onto. Suppose we have $x_1, x_2 \in Q$ such that $r_q(x_1) = r_q(x_2)$. It follows that $x_1 * q = x_2 * q$ and by right cancellation law, we have $(x_1 * q)/q = (x_2 * q)/q \Rightarrow x_1 = x_2$. Thus, we know that r_q is also one-to-one. Therefore, r_q is a isomorphism from Q to itself, so r_q is an automorphism. □

Definition 3.1.3. An **inner automorphism** of q in quandle Q, is the automorphism r_q defined in Theorem 3.1.2. The group generated by all inner automorphisms of Q is called the **inner automorphism group** of Q, and is denoted by $Inn(Q)$. △

Note that not every element in the inner automorphism group is a right translation.

Definition 3.1.4. The **orbit** of an element $x \in Q$ is $\{r_q(x) \in Q \mid q \in Q\}$, and is denoted by $Orb(x)$. △

The orbit of x is the set of all possible elements of Q that can be reached by repeatedly acting on x with other elements of Q in any combination. In other words, y is in the orbit of x if and only if $y = (\cdots((x * q_1) * q_2) \cdots * q_{n-1}) * q_n$ for some $q_1, q_2 \ldots, q_n \in Q$.

Corollary 3.1.5. *The orbits of a quandle Q are precisely the connected components of its right Cayley graph. Thus, Q is connected if it has only 1 orbit, in which case $Inn(Q)$ acts transitively on Q.*

3.2 Cycle Structures

From Definition 3.1.1 we know that each element a in a quandle Q defines an inner automorphism by right action. In other words, the element a defines a permutation on the quandle elements of Q, and this property can be reflected in Q's multiplication table.

Definition 3.2.1. Let $\sigma = c_1\, c_2 \ldots c_n$ be a permutation, where $c_1, c_2 \ldots c_n$ are disjoint cycles. Let l_i be the length of c_i for $1 \le i \le n$, and assume that $l_1 \le l_2 \le l_3 \ldots \le l_n$. The **cycle structure** for σ is the n-tuple vector $(l_1\, l_2 \ldots l_n)$. △

Example 3.2.2. Consider the linear quandle $LQ(5,2)$ in Example 2.2.8 with the following multiplication table:

*	0	1	2	3	4
0	0	4	3	2	1
1	2	1	0	4	3
2	4	3	2	1	0
3	1	0	4	3	2
4	3	2	1	0	4

Note that the first column defines a permutation on the quandle elements: 02413. We can decompose this permutation into 2 disjoint cycles: (0)(1 2 4 3). We say that the first column has cycle structure (1 4) based on the length of each cycle. ♢

Theorem 3.2.3. *Let Q be a quandle such that $Aut(Q)$ acts transitively on Q. Then all of the columns in Q have the same cycle structure.*

Proof. Let $q \in Q$, r_q be the inner automorphism associated with q, and $\pi \in Aut(Q)$. Then we know that r_q, π, and the conjugate $\pi \circ r_q \circ \pi^{-1}$ are all permutations on the elements of Q. Then we have

$$(\pi \circ r_q \circ \pi^{-1})(x) = \pi(r_q(\pi^{-1}(x))) \tag{3.2.1}$$

$$= \pi(\pi^{-1}(x) * q) \tag{3.2.2}$$

$$= x * \pi(q) = r_{\pi(q)}(x) \tag{3.2.3}$$

By Proposition 9.20 in Humphreys [4], we know that $\pi \circ r_q \circ \pi^{-1}$ has the same cycle structure as r_q. Since $Aut(Q)$ is transitive, we know that for every pair of $q_1, q_2 \in Q$, there exists some π_0 such that $\pi_0(q_1) = q_2$. It follows from Equation 3.2.3 that the r_q's have the same cycle structure for all $q \in Q$.

Furthermore, we know that each column in Q is a permutation of some r_q, so all of the columns have the same cycle structure.

□

Corollary 3.2.4. *All of the columns of a connected quandle have the same cycle structure.*

Proof. Let Q be a connected quandle. By Corollary 3.1.5, we know that $Inn(Q)$ acts transitively on Q. Since $Inn(Q) \subset Aut(Q)$, we know that $Aut(Q)$ acts transitively on Q. By Theorem 3.2.3, we know that all of the columns of Q have the same cycle structure. □

With this property of cycle structures, we can define a characteristic for all the connected quandles and use it as a computational invariant.

Definition 3.2.5. Let Q be a connected quandle. The **permutation signature** of Q is the cycle structure of its columns. △

With permutation signature, we can distinguish some of the isomorphism classes among connected quandles without directly using `isofilter`.

Example 3.2.6. Recall the dihedral quandles in Example 2.2.7. $Dih(5)$ is another connected quandle of size 5. It has the following multiplication table:

*	0	1	2	3	4
0	0	2	4	1	3
1	4	1	3	0	2
2	3	0	2	4	1
3	2	4	1	3	0
4	1	3	0	2	4

The permutation of the first column can be broken down to disjoint cycles (0) (1 4) (2 3), and thus this quandle has permutation signature (1 2 2). By comparing this with the permutation signature (1 4) of $LQ(5,2)$, we know that $Dih(5)$ and $LQ(5,2)$ are distinct although they are both connected quandles of size 5. ◊

However, permutation signature cannot comprehensively distinguish any two connected quandles of the same size. In the later chapter, we will introduce a stronger invariant.

Example 3.2.7. Another connected quandle of size 5 would be the linear quandle $LQ(5,3)$, and it has the following multiplication table:

*	0	1	2	3	4
0	0	3	1	4	2
1	3	1	4	2	0
2	1	4	2	0	3
3	4	2	0	3	1
4	2	0	3	1	4

The first column can be decomposed as a product of two disjoint cycles: (0) and (1 3 4 2), so this quandle has the same permutation signature as $LQ(5,2)$ in Example 3.2.2. We would not be able to distinguish these two isomorphism classes just by permutation signature. In Example 4.1.6, we will use a different invariant to distinguish these two isomorphism classes. ◊

3.3 Algorithm to Compute Permutation Signature

Computing the permutation signature for a connected quandle is essentially computing the cycle structure for its first column. Thus, this algorithm boils down to computing the cycle structure of a permutation.

Our problem can be rephrased as the following: given an array of the integers containing $\{1 \dots n\}$, how do we get the cycle structure from the ordering? The first procedure we need to do is to initialize some cycle leader (as the first element in a cycle).

Algorithm 3.3.1.

```
2: procedure CycleLeader($array[1 \dots n], start$)
       $index \leftarrow start$                    ▷ $start$ is the starting array index to search
```

```
    **while** $index < n$ and $array[index]$ is already in a cycle **do**
        $index \leftarrow index + 1$ ▷ Iterate until we find we find an element not in a cycle yet
    **end while**
    **if** $index = n$ **then**   **return** $-1$                ▷ Indicate no more cycle
    **else**   **return** $index$                        ▷ The new index to start a cycle
    **end if**
**end procedure**
```

After finding cycle leader l, iterating its cycle is just contiguously applying the permutation π on l until we have $\pi^k(l) = l$. The following algorithm will output a list containing all such k's. Notice that we are using another array of the same size *Labels* to keep track of whether an element is already contained in a cycle.

Algorithm 3.3.2.

```
**procedure** CycleStruct($Perm[1 \dots n]$)
    Initialize array $Labels[1 \dots n]$ such that it contains all 1's
    $Labels[1] \leftarrow 0$                    ▷ The first element always forms a trivial cycle
    $start \leftarrow 2,\ OutList \leftarrow$ an empty list
    **while** $(index \leftarrow \text{CycleLeader}(Perm[], start)) \neq -1$ **do** ▷ There is still more cycle
        $count \leftarrow 1$                     ▷ Keep tracks of the current cycle length
        $next \leftarrow Perm[index]$
        $Labels[next] \leftarrow 0$               ▷ Label this element as included in a cycle
        **while** $next \neq index$ **do**                        ▷ Keep looping the cycle
            $next \leftarrow array[next]$              ▷ Look at the next element in the cycle
            $Labels[next] \leftarrow 0$ , $count \leftarrow count + 1$
        **end while**
        append $count$ to $OutList$
        $start \leftarrow start + 1$
        end while
16: return $OutList$
    end procedure
```

This algorithm is essentially telling us how to iterate through the elements $\{1, 2 \ldots n\}$ based on the permutation array and the label array. We also know that the checking on these two arrays takes constant time. Thus, this procedure should have complexity $\Theta(n)$. The actual implementation is written in C, and is included in the appendix.

4
Quandle Terms

In this chapter, we will introduce quandle identities as another computable invariant for quandles.

4.1 Quandle Terms and Identities

Definition 4.1.1. A **term** of n variables $\{x_1, x_2 \ldots, x_n\}$ for a quandle Q can be recursively defined as follows

- The expressions $x_1, x_2, \ldots x_n$ are terms;
- If t_1 and t_2 are terms, then $t_1 * t_2$ and t_1/t_2 are both terms.

△

Example 4.1.2. $x * y$, $(x * y) * y$, and $(x * y)/x$ are examples of terms for quandles. For dihedral quandle of size 5, we can translate a quandle term to a term in the cyclic group of size 5. For example, $x * y = 2y - x \mod 5$ and $(x * y) * y = 2x - (2y - x) = 3x - 2y \mod 5$.

Moreover, we can view each term of n variables as a map from $\underbrace{Q \times Q \cdots \times Q}_{n \text{ times}}$ to Q. ◊

In this senior project, I am mostly using terms of two variables although some of the work can be extended by looking at terms of more variables.

Definition 4.1.3. A **quandle identity** is an expression of the form $t_1(x_1, x_2 \dots x_n) = t_2(x_1, x_2 \dots x_n)$ in which both t_1 and t_2 are quandle terms. $\triangle$

Example 4.1.4. Many interesting types of quandles are defined by identities. The medial quandles introduced in Definition 2.1.6 are defined by the identity $(a * b) * (c * d) = (a * c) * (b * d)$. The involutory quandles in Definition 2.1.7 are defined by the identity $(x * y) * y = x$. Note that this identity can also be written as $r_y^2(x) = x$ in which r_y is the inner automorphism associated with y. This implies that the permutation for r_y only contains 1-cycles and 2-cycles. $\lozenge$

Definition 4.1.5. Let $t_1 = t_2$ be an identity and Q be a quandle. We say Q **models** $t_1 = t_2$ if Q satisfies the identity. Two identities are **equivalent** if they are provably the same in the pure equational theory of quandles. $\triangle$

Identities can be very useful invariants because isomorphic quandles satisfy the same identities.

Example 4.1.6. Recall that in Example 3.2.7, both of the quandles $LQ(5, 3)$ and $LQ(5, 2)$ have the same permutation signature. However, we can actually distinguish these two isomorphism classes using the identity $(x/y) * x = y$. This is because $LQ(5, 2)$ models the identity while $LQ(5, 3)$ does not. $\lozenge$

4.2 Normal Forms

Because of right cancellation of the quandle axioms, each 2-variable quandle identity is equivalent to an identity in the form of $t(x, y) = x$ or $t(x, y) = y$, so that there is only one variable on the right hand side. For example, consider the identity $(x * y) * y = (x/y) * x$, which can be rewritten in two steps:

$$((x * y) * y)/x = x/y$$

$$(((x * y) * y)/x) * y = x$$

Furthermore, we can rewrite each term into **normal form** using the rewrite system introduced by Hannah Quay-De La Vallee [7] in her senior project. The normal form rewriting rules are listed below, and the goal of the system is to rewrite a term into the form: $(\cdots((x_{i1}\, o_1\, x_{i_2})\, o_2\, x_{i_3})\cdots x_{i_{n-1}})\, o_{n-1}\, x_n$ in which $o_k \in \{*, /\}$. This term also has to satisfy the following

- $i_1 \neq i_2$
- For all $1 \leq k \leq n-2$, we have $o_k \neq o_{k+1}$ implies $i_{k+1} \neq i_{k+2}$.

Note that in the following rules each of the variables X, Y, Z represents a quandle term.

1. $X * X \to X$ and $X/X \to X$
2. $(X * Y)/Y \to X$ and $(X/Y) * Y \to X$
3. $X * (Y * Z) \to ((X/Z) * Y) * Z$
4. $X * (Y/Z) \to ((X * Z) * Y)/Z$
5. $X/(Y * Z) \to ((X/Z)/Y) * Z$
6. $X/(Y/Z) \to ((X * Z)/Y)/Z$

Definition 4.2.1. A **normal form term** is a quandle term such that we cannot further reduce it with any of the rewriting rules. A **normal form identity** is an identity of the form $t(x_1, x_2 \ldots x_n) = x_i$ such that $1 \leq i \leq n$ and t is a normal form term. $\triangle$

Proposition 4.2.2. *Every quandle identity $t_1 = t_2$ is equivalent to a normal form identity.*

Proof. Based on the right cancellation rule and the term rewriting system, we can turn every quandle identity $t_1 = t_2$ into a normal form identity: first rewrite the terms t_1 and t_2 into normal form terms t'_1 and t'_2 respectively; use the right cancellation rule to remove all of the variables in t'_2 except the leftmost one and append them to t'_1. □

Example 4.2.3. Consider the identity $x * (x * y) = y * (y/x)$. We can rewrite it with the following steps:

$$\begin{aligned} \text{normal form terms on both sides: } &\to ((x/y) * x) * y = ((y * x) * y)/x \\ \text{reducing variables on the right: } &\to (((x/y) * x) * y) * x = (y * x) * y \\ &\to ((((x/y) * x) * y) * x)/y = y * x \\ &\to (((((x/y) * x) * y) * x)/y)/x = y \end{aligned}$$

◊

Thus, every identity is equivalent to a normal form identity, and we can use the normal form identity to represent each identity equivalence class.

Definition 4.2.4. Let t be a term of normal form, the **size** of t is the number of operations in t. Similarly, the size of the normal form identity $t = x_i$ is the size of t. △

Theorem 4.2.5. *Let $T_{n,m}$ be the set of all normal form terms of n variables and size m. Then $|T_n| = 2n(n-1)(2n-1)^{m-2}$.*

Proof. Let $t \in T_{n,m}$. Then we can write t as a sequence of variables and operations $(v_1, o_1, v_2, o_2 \dots o_m, v_{m+1})$ in which v_i's are variables and o_i's are the two binary operations for quandles. By Definition 4.2.1 and the rewrite system, we know that t satisfies the following:

- v_1 and v_2 are different variables;

- For $i \geq 2$, v_i and v_{i+1} must be different variables if o_{i-1} and o_i are different operations.

For the subsequence $\{v_1, o_1, v_2\}$, there are $2n(n-1)$ different combinations. In the following subsequences $\{o_i, v_{i+1}\}$ with $i \geq 2$, we would have either v_i and v_{i+1} are different variables or o_i and o_{i-1} are the same operation. This means at each step we have $2n-1$ ways to append an operation along with a variable. For a total number of m operations, we would have $2n(n-1)(2n-1)^{m-1}$ ways to form such a term t. □

Proposition 4.2.6. *Testing a normal form identity of size m with k variables on a quandle of size n has complexity $O(m\,n^k)$.*

Proof. Let $t = v$ be a normal form identity of size m with k variables. For a quandle with n elements, there are n^k ways to substitute the variables in the identity. For each substitution, computing the value of t takes m look-up's in the multiplication table because it has m operations. Overall, it takes at most $m \times n^k$ steps to verify that the quandle models the identity, so this procedure has complexity $O(m\,n^k)$. □

4.3 Generating Normal Form Identities

Generating a normal form identity of size m is basically generating a normal form term and appending a correct variable on the other side. Note that each n-variable normal form term t corresponds to exactly $n-1$ normal form identities. This is because for each normal form identity $t = x_i$, the variable x_i might as well differ from the rightmost variable in t. For example, the identity of size 3: $(x * y) * x = x$ would be reduced to an identity of size 2: $x * y = x$ by the right cancellation law. In this work, we only use 2-variable normal form identities, so each normal term t corresponds to exactly one normal identity.

Since we are using identities to test on quandles, so we would want to avoid generating equivalent identities. Although it is hard to determine the equivalence classes for identities, we can produce equivalent identities by using the right cancellation law and permuting the variables. In the actual implementation, we would avoid test these equivalent identities.

Example 4.3.1. Consider the identity $(((x*y)/x)/x)*y = x$. We have

$$\begin{aligned}&\Leftrightarrow ((x*y)/x)/x = x/y\\ &\Leftrightarrow (x*y)/x = (x/y)*x\\ &\Leftrightarrow x*y = ((x/y)*x)*x\\ &\Leftrightarrow x = (((x/y)*x)*x)/y\end{aligned}$$

Thus, the identity $(((x*y)/x)/x)*y = x$ is equivalent to the identity $(((x/y)*x)*x)/y = x$ although their corresponding normal terms are not the same. We say that $(((x*y)/x)/x)*y$ is the **reverse term** of $(((x/y)*x)*x)/y = x$.

By switching the two variables we get the identity $(((y*x)/y)/y)*x = y$, which is also equivalent to the original identity. ◊

By the proof of Theorem 4.2.5, we know that each normal form identity of size $n-1$ can be extended to 3 normal form identities of size n because we are using 2 variables. For example, the identity $x*y = x$ can be extended to $(x*y)*y = x$, $(x*y)*x = y$, and $(x*y)/x = y$. Then we have an algorithm for generating identities recursively.

With this inductive property, we can generate normal form terms quite easily using Prolog. The following code allows us to generate normal form terms for all sizes. It avoids producing the redundant terms for permutations of the variables by restricting the leftmost variable to be `x`. Note that the predicate `term(T, N)` means `T` is a normal form term of size `N`. The predicate `variable(Z)` just means that `Z` is either `x` or `y`.

```
    %base case: term of size 1
term(star(x,y), 1).
term(div(x,y), 1).
    %case1: We can append both x and y to form a new term
```

```
term(star(star(E, x), Z), M):- M>1, N is M - 1, variable(Z),
                               term(star(E,x), N).
term(div(div(E,x),Z), M)    :- M>1, N is M - 1, variable(Z),
                               term(div(E,x), N).
term(star(star(E, y), Z), M):- M>1, N is M - 1, variable(Z),
                               term(star(E,y), N).
term(div(div(E, y), Z), M)  :- M>1, N is M - 1, variable(Z),
                               term(div(E,y), N).
    %case2: We can only append the opposite variable to form a new term
term(star(div(E, x), y), M) :-  M>1, N is M - 1, term(div(E,x), N).
term(div(star(E, x), y), M) :-  M>1, N is M - 1, term(star(E,x), N).
term(star(div(E, y), x), M) :-  M>1, N is M - 1, term(div(E,y), N).
term(div(star(E, y), x), M) :-  M>1, N is M - 1, term(star(E,y), N).
```

For each normal form term `T`, we can generate an identity using predicate `identity(T, V)`, which represents `T = V`. The predicate `switch(X, Y)` ensures that the variables `X` and `Y` are different, and hence let us append the correct variable. The Prolog code is as follows:

```
term(star(_, X), Y)   :- switch(X,Y).
term(divide(_, X), Y) :- switch(X,Y).
```

We also have another predicate `reverseTerm(T1, T2)`, which represents the relation that `T1` and `T2` are mutually reverse terms as defined in Example 4.3.1. With this predicate, we can remove all of the equivalent duplicates from a list of normal form identities (represented by their normal form terms):

```
removeReverse([], []).
removeReverse([H|T], L)    :- reverseTerm(H, T), removeReverse(T, L).
removeReverse([H|T], [H|L]):- not(reverseTerm(H, T)), removeReverse(T, L).
```

As shown above, the predicate `removeReverse(L1, L2)` means `L2` is the list we obtain from `L1` by removing its reverse terms.

5

Filtering Quandles

In this chapter, we will introduce an augmented algorithm for filtering isomorphic quandles. In this algorithm, we take advantage of the invariant checking introduced in the previous chapters to improve the original program `isofilter`. Our implementation mainly integrates tools from LADR, Prolog, and GAP4. This chapter is also intended to be a manual for future usage and extension.

5.1 An Augmented Algorithm to Filter Quandles

According to Birrell [1], we know that determining whether two quandles are isomorphic is computationally intractable. There is no currently known algorithm for directly determining whether two quandles are isomorphic in polynomial time. Indeed, the `isofilter` program within LADR checks the possible isomorphisms between first-order models, and it runs very slowly for large quandles. In order to improve the speed for filtering isomorphic quandles, we should avoid directly applying `isofilter` on quandles.

Compared to `isofilter`'s algorithm, invariants checking has a much better run time. By Proposition 4.2.6, we know that testing a quandle of size n on an identity of size m

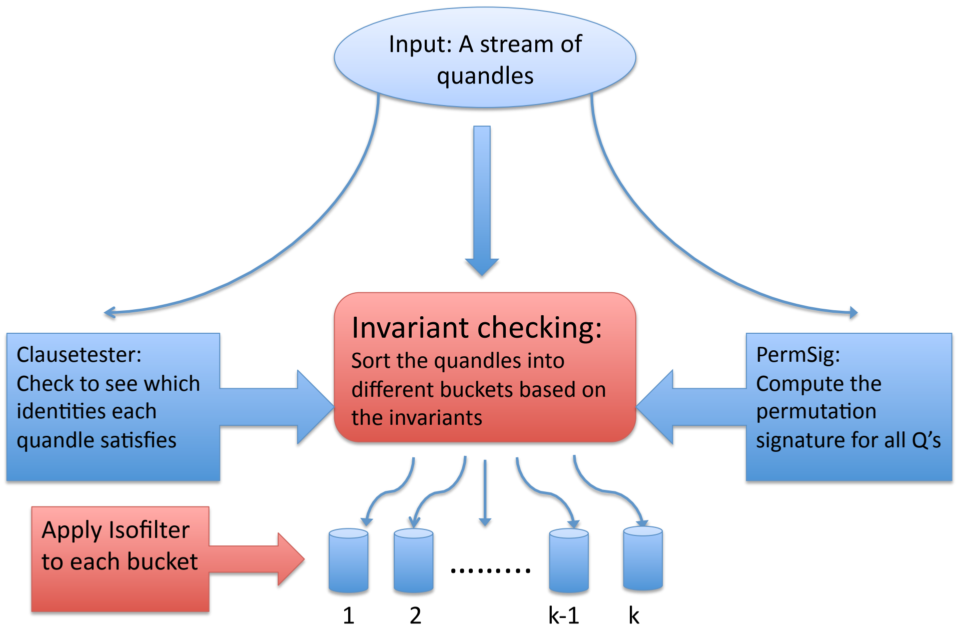
Input: A stream of quandles
Invariant checking:
Sort the quandles into different buckets based on the invariants
Clausetester:
Check to see which identities each quandle satisfies
PermSig:
Compute the permutation signature for all Q's
Apply Isofilter to each bucket
....
1
2
k-1
k

Figure 5.1.1. Filtering Quandles with Invariants

takes $O(m\,n^2)$ time and Algorithm 3.3.2 for computing permutation signature has run time $\Theta(n)$. Thus, it is a good strategy to first compute these invariants for the quandles and partition them into different buckets such that the quandles in each bucket share the same invariants. Then we can largely reduce the number of `isofilter` checkings since we just need to apply it to each bucket rather than the entire list of quandles.

This augmented algorithm to filter a stream of quandles is illustrated in Figure 5.1.1. In this chapter, we will discuss the details of each component of the algorithm along with the computational tools in the actual implementation.

5.2 Computational Tools

5.2.1 Interpretation format

The LADR provides very useful tools for studying first-order models. In particular, every first-order model is declared as an interpretation and represented in a `interp` format. For example, the linear quandle $LQ(5,3)$ will be written in a `interpretation` clause:

```
interpretation(5, [number=3, seconds=0], [
function(*(_,_), [0, 3, 1, 4, 2,
                  3, 1, 4, 2, 0,
                  1, 4, 2, 0, 3,
                  4, 2, 0, 3, 1,
                  2, 0, 3, 1, 4]),
function(/(_,_), [0, 4, 3, 2, 1,
                  2, 1, 0, 4, 3,
                  4, 3, 2, 1, 0,
                  1, 0, 4, 3, 2,
                  3, 2, 1, 0, 4])]).
```

A `interp` format file is a file containing multiple `interpretation` clauses separated by spaces. Notice this quandle is represented by two multiplication tables associated to the two binary operations. In particular, each table is represented by a clause `function(O, L)`, in which `O` specifies the operation and the list `L` gives the multiplication table in one dimension. In order to work with the LADR, we need to rewrite every quandle into such a `interp` format. In our actual implementation, we also need to produce the multiplication table for the operation / from the table of $*$.

5.2.2 Isofilter

As a program inside of the LADR, `isofilter` reads in `interp` format file and removes isomorphic duplicates. For example, the following command line

```
./isofilter < BA2.interps > BA2.interps2
```

would filter the isomorphic duplicates in `BA2.interps` and output only one model for each isomorphism class to the file `BA.2interps2`. At the end of the output file, `isofilter` would also print out information like the following.

```
% isofilter: input=10, kept=2, checks=8, perms=14, 0.02 seconds.
```

The `input` counts the number of input models, and `kept` counts the number of isomorphism classes among them. The number of `checks` counts the number of `isofilter` checkings on pairs of models, and the value of `perms` counts the total number of permutations the program has searched. It is observed that the run-time of isofilter (as shown at the end of the info) is proportional to the value of `perms`.

The algorithm of `isofilter` has the following structure.

Algorithm 5.2.1.

```
 2: procedure ISOFILTER(A list of models $QList$)
        $OutList$ initialized to be an empty list
 4:     $Q \leftarrow QList.next$                          ▷ The first model in the list
        Insert $Q$ into $OutList$
 6:     while $(Q \leftarrow QList.next) \neq$ null do
            if $Q$ is not isomorphic to any model in $OutList$ then Append $Q$ to $OutList$
 8:         end if  $M \rightarrow$ the next element in $QList$
        end while
10: return $OutList$
    end procedure
```

However, the isomorphism checking in line 5 is a very expensive computation. For checking whether model A is isomorphic to B, the program would search for possible isomorphisms from A to B. If A and B are non-isomorphic models of size m, `isofilter` would search through all possible isomorphisms and answers that they belong to different isomorphism classes (in this case, the value of `perms` is mostly $m!$). Thus, the `isofilter` tends to produce an answer faster when filtering two models that are actually isomorphic because the enumeration of permutations would stop when an isomorphism is found. We can verify this claim from some empirical results of using `isofilter`. For example, when we apply `isofilter` to an `interp` file with 300 size-9 quandles that are all isomorphic the quandle $Dih(3) \times Dih(3)$, it takes about 0.21 seconds to determine all of them are isomorphic. Then, for an `interp` file containing 3 quandles of size 9 belonging to 3 different isomorphism classes, it takes `isofilter` 0.22 seconds to process.

Therefore, invariants checking would largely reduce isomorphism searching among non-isomorphic quandles because it would separate most of non-isomorphic quandles at an early stage.

5.2.3 Clausetester

As another program of the LADR, the program `clausetester` tests a stream of identities on a set of models. In our project, we modified the source code of `clausetester.c` to generate a program for invariants checking.

For each input identity, `clausetester` will print out the interpretations that model the identity. The output will also contain the number of identities each interpretation models. A command line for using `clausetester` would look like:

```
clausetester quandles.interps < size2.identities > quandles.out
```

In this example, `clausetester` would test the identities in `size2.identities` on the interpretations in the file `quandles.interps` and output the results to the file quandles.out. We need to properly label the identities in size2.identities, and it should look like:

```
(x * y) * y = y #label(1)
(x * y) / x = y #label(2)
(x / y) * y = x #label(3)
```

And the output file quandles.out would contain information about the identities and the models.

```
(x * y) / x = y #label(2) 1 2 5
(x / y) * y = x #label(3) 2 5
%interp 1 models 8 of 15 clauses
```

In order to tailor the use of clausetester to our identities checking on quandles, we modified the source code of clausetester.c, so it will output information as Prolog predicates. The output above would be converted into the following.

```
identitySatisfy(1, 2).  %means quandle 1 models identity 2
identitySatisfy(2, 2).
identitySatisfy(5, 2).
identitySatisfy(2, 3).
identitySatisfy(5, 3).
numSatisfy(1, 8).       %means quandle 1 models 8 identities
```

Thus, the output is more directly expressed as relations between the quandles and identities. Every time we call the program clausetester, we actually generate a Prolog database of quandles and identities.

Furthermore, we also add the permutation signature computation inside of clausetester to make the invariant checking more efficient. In addition to the identities checking output, our version of clausetester would also produce predicates representing the permutation signature of each quandle. Such a predicate would look like the following. Notice that the we do not include the trivial cycle for simplicity.

```
permSig(1,[2,2,3])   %means quandle 1 has permutation signature (2 2 3)
```

Therefore, this augmented version of `clausetester` becomes a program for quandle invariants checking. The output is a Prolog database of input quandles and their relations with the invariants.

5.2.4 Filtering Quandles with Prolog

As a general purpose logical programming language, Prolog is consistently used in our work. Unlike procedure programming languages such as C and Java, the program logic is expressed in terms of relations. With the database generated by `clausetester`, we can easily determine whether two quandles are possibly isomorphic with the following code.

```
iso(A, B):- quandle(A), quandle(B), A < B,
            numSatisfy(A, N), numSatisfy(B, N),
            bagof(I1, identitySatisfy(A, I1), L),
            bagof(I2, identitySatisfy(B, I2), L),
            permSig(A, P1), permSig(B, P2),
            mergesort(P1,P), mergesort(P2, P).
```

The predicate `iso(A, B)` means that quandles `A` and `B` share the same invariants including the set of identities they satisfy and their permutation signatures. Based on the `iso` relation, we can place the quandles satisfying the same invariants in a bucket and send them to `isofilter` to see whether they are actually isomorphic. Here is the `bucket` predicate.

```
bucket([A|T]) :- bagof(B, iso(A,B), T),
                 retract(quandle(A)), retractQuandles(T).
```

Note that the `retract` and `retractQuandles` remove quandles from the database once they are in a bucket, so it avoids producing redundant buckets. We also avoid computing the buckets with only one quandle because the goal `bagof(B,iso(A,B),T)` will not be satisfied if quandle `A` is identified as non-isomorphic to all others. Thus, we are only sending the non-trivial buckets to `isofilter`. In this step, we are writing each bucket of quandles to a `interp` format file, and then call the program `isofilter` to process it.

All of the steps including invariant checking, partitioning quandles into buckets and applying `isofilter` to each bucket can be done with just one command line shown below. In particular, `quandles.interp` is the file for input quandles, `identities` is the file containing certain normal form identities, and `invariantChecker` is our augmented `clausetester` that does invariant checking. The program `FilterQuandles` is an Prolog executable file that sorts quandles into different buckets and send them to `isofilter`.

```
./invariantChecker quandles.interp < identities |./FilterQuandles
```

And the output in `stdout` is in the following format:

```
There are 42 Quandles.
There are 354 Identities.
Buckets cleaned
The following buckets have sizes more than 1
[2,3]
Sent to file: Bucket1
[11,20]
Sent to file: Bucket2
[12,25]
Sent to file: Bucket3
[13,22,28,39]
Sent to file: Bucket4
[14,15,31,33]
```

```
Sent to file: Bucket5
[16,30,32]
Sent to file: Bucket6
[19,38]
Sent to file: Bucket7
[21,27]
Sent to file: Bucket8
Applying Isofilter to each bucket. Might take forever! Feel free to interrupt
isofiltering Bucket8...
isofiltering Bucket7...
isofiltering Bucket6...
isofiltering Bucket5...
isofiltering Bucket4...
isofiltering Bucket3...
isofiltering Bucket2...
isofiltering Bucket1...
****Done!****
```

If we run the command for a set of large quandles, isofiltering each bucket can still take a long time even if the bucket has a small size.

6

Conclusion

In this chapter, we will analyze the performance of our algorithm and also propose some potential improvements.

6.1 Analysis

6.1.1 Minimize the Use of Isofilter

Since our invariant checking has a much better run-time performance than `isofilter` in distinguishing non-isomorphic quandles, our quandle filtering algorithm would work more efficiently if we can reduce the use of `isofilter`. Consider a stream of n connected quandles of the same size m (quandles of different sizes are certainly not isomorphic), there are two cases in which we do not need to apply `isofilter` at all:

1. We can separate all of the quandles into exactly n different buckets;

2. We know in advance there are k isomorphism classes for connected quandles of size m, and we know the invariants that can distinguish all of them.

While the first case depends on the input quandles, the second case depends on how good the invariants are in terms of distinguishing the isomorphism classes for size-m quandles. From the GAP4 quandle library provided by Professor James Belk, we obtain a comprehensive list of isomorphism classes for connected quandles up to size 30. When we test our invariant checking on the isomorphism classes of each size, it turns out that we can distinguish most of the classes. By using normal form identities up to size 12 and the quandles' permutation signature, we are able to distinguish 294 among all of the 359 isomorphism classes. The other 65 isomorphism classes are placed in 19 different buckets. While most of the buckets contain 2 or 3 quandles, we encounter an unusually large bucket for size-27 quandles, which contains 19 quandles. It is possible that many of the quandles in this bucket might actually model the same identities since they are subquandles of a conjugacy quandle.

If we can distinguish all of the isomorphism classes for size m just by invariant checking, we would have a dramatic speed improvement on the quandle filtering process. For example, it takes `isofilter` more than 18 hours to distinguish all of the 11 isomorphism classes for connected quandles of size 13. However, if we apply our augmented algorithm to the same set of quandles, it would only take less than a second. Since we can distinguish all of the isomorphism classes for size 13 with five identities and their permutation signature, `isofilter` is not used at all. So far we know that our invariants can distinguish all of the isomorphism classes for connected quandles up to size 30 except for sizes 12, 15, 21, 24, 27, 28, and 30.

Even if we cannot fully distinguish all of the isomorphism classes for size-m quandles, we can still have a much better run-time. For example, among the 10 isomorphism classes for size-12 connected quandles, two of them are not distinguishable by our invariants. Then we would need to apply `isofilter` to the bucket containing these two quandles.

The filtering process in total would take around 55 seconds, whereas directly applying `isofilter` to the same set of quandles would take more than an hour to finish.

Notice that in the two examples above, we are only testing our program on a set of quandles that are pairwise non-isomorphic. Although in practice we would be dealing with a set of random quandles as input, we know that most of the computation would be spent on applying `isofilter` on non-isomorphic quandles as shown in Section 5.2.2. In the end, the performance is mostly determined by the number of isomorphism classes contained in each bucket.

6.1.2 Selecting Identities

Although we can keep computing new identities of larger sizes to strengthen the invariant checking, but the number of normal identities grows exponentially in number of operations by Theorem 4.2.5. Computing the entire list of normal form identities becomes expensive for large sizes. For example, there are more than a million normal form identities of size 12, so it takes a long time to compute a comprehensive list of such size or larger.

It is also not practical to test on all of the normal form identities we generate. Thus, we would like to reduce the number of redundant identities. The following theorem would tell us the lower bound for the identities we need.

Theorem 6.1.1. *If we can fully distinguish n pairwise quandle isomorphism classes just using m identities, then we must have $m \geq \lg n$.*

Proof. Let Q be a quandle. Then we can convert Q to a m-tuple binary vector $(v_1\ v_2 \dots v_m)$ such that each $v_i = 1$ if Q models the i-th identity, and $v_i = 0$ otherwise. For m identities, we can form 2^m different vectors. If two quandles have two different binary vectors, then we can distinguish them. In order to distinguish all of the n quandle isomorphism classes just using the identities, we must have $n \leq 2^m$ by pigeonhole principle. Thus, we have $m \geq \lg n$. $\square$

Since $\lg 359 \approx 8.48$, ideally we would be able to filter all of the connected quandles up to size 30 just using 9 identities. However, searching for such identities for all isomorphism classes is not practical because some quandles might model exactly the same identities. Recall the transposition quandles T_n we introduce in Example 2.2.2. It turns out that we cannot distinguish $T_4 \times T_4$ and T_9 using 2-variable identities. We can show this by first proving the following theorem.

Theorem 6.1.2. *Let integer $n \geq 4$ and $I(Q)$ denote all of the identities over 2 variables that a quandle Q models. Then $I(T_n) = I(T_3) \cap I(U_2)$.*

Proof. Let $x, y \in T_n$ such that $x \neq y$. Since x, y are also transpositions in S_n, we can write $x = (a\ b)$ and $y = (c\ d)$ in which $a, b, c, d \in \{1, 2, \ldots n\}$. If $(a\ b)$ and $(c\ d)$ are disjoint, we then have

$$x * y = y^{-1}xy = (c\ d)(a\ b)(c\ d) = (a\ b) = x$$
$$y * x = x^{-1}yx = (a\ b)(c\ d)(a\ b) = (c\ d) = y$$

We then know that the subquandle generated by x, y is U_2.

If $(a\ b)$ and $(c\ d)$ are not disjoint, we then can assume that $b = c$. We then have

$$x * y = y^{-1}xy = (c\ d)(a\ b)(c\ d) = (a\ d)$$
$$y * x = x^{-1}yx = (a\ b)(c\ d)(a\ b) = (a\ d)$$

Thus, we know that the subquandle generated by x, y is T_3.

Therefore, the subquandles generated by two elements in T_n are either T_3 or U_2. Then an identity that T_3 models must also be satisfied by both U_2 and T_3. Thus, we must have $I(T_n) \supset I(T_3) \cap (U_2)$.

Let $t_1 = t_2$ be an identity of 2 variables that T_n models. This means that for every pair of $x, y \in T_n$, we would have $t_1(x, y) = t_2(x, y)$. Since x, y can possibly generate T_3 or U_2, this identity would be in the set $I(T_3) \cap I(U_2)$. This means $I(T_n) \subset I(T_3) \cap I(U_2)$.

Hence, we show that $I(T_n) = I(T_3) \cap I(U_2)$. $\square$

Proposition 6.1.3. *Let Q, R be two quandles. Then $I(Q \times R) = I(Q) \cap I(R)$.*

Proof. First, every identity modelled by the quandle $Q \times R$ must also be satisfied by both components Q and R. Thus, $I(Q \times R) \subset I(Q) \cap I(R)$.

Let $t_1 = t_2$ be an identity in $I(Q) \cap I(R)$. Then for every substitution $(q_1, r_1), (q_2, r_2) \in Q \times R$, we would have

$$\begin{aligned} t_1((q_1, r_1), (q_2, r_2)) &= (t_1(q_1, q_2), t_1(r_1, r_2)) \\ &= (t_2(q_1, q_2), t_2(r_1, r_2)) = t_2((q_1, r_1), (q_2, r_2)) \end{aligned}$$

Therefore, we know that $Q \times R$ also models this identity, so $I(Q \times R) \supset I(Q) \cap I(R)$. Hence, we have $I(Q \times R) = I(Q) \cap I(R)$. □

The orders of $T_4 \times T_4$ and T_9 are the same because $|T_4 \times T_4| = \left(\frac{4 \times 3}{2}\right)^2 = 36 = \frac{9 \times 8}{2} = |T_9|$. By Theorem 6.1.2 and Proposition 6.1.3, we know that $I(T_4 \times T_4) = I(T_9) = I(T_3) \cap I(U_2)$. Thus, we would not be able to distinguish these two quandles of the same size just by using 2-variable identities.

However, our invariant checking would still be able to distinguish these two quandles because they have different permutation signatures. The permutation signature of $T_4 \times T_4$ contains sixteen 2's, whereas the one of T_9 only contains seven 2's.

We also know that some identities can reflect the permutation signatures.

Proposition 6.1.4. *Let Q be a connected quandle that models the identity, and $(p_1\ p_2 \ldots p_k)$ be the permutation signature of Q. Then Q models the identity $((x * \underbrace{y) * y \cdots y) * y}_{n} = x$ if and only if $p_i | n$ for $1 \leq i \leq k$.*

Proof. Let $p, q \in Q$ and consider the inner automorphism r_q. Since r_q is a permutation on the elements of Q, we can decompose it into disjoint cycles. Let the length of the cycle containing p be l.

Suppose Q models the identity, then we have $((p * \underbrace{q) * q \cdots q) * q}_{n} = x$. It follows that applying r_q on p for n times would give back p. This means n is divisible by l. Also, we know that each p_i in the permutation signature of Q is the length of some cycle in r_q, so each p_i also divides n.

Suppose that $p_i | n$ for $1 \leq i \leq k$. Then for each i, we can write $n = p_i\, m_i$ for some natural number m_i. Let $x, y \in Q$. Then we can say x belongs to a cycle in r_y that has length p_j. It follows that $r_y^n(x) = r_y^{p_j\, m_j}(x) = x$. Thus, the identity $((x * \underbrace{y) * y \cdots y) * y}_{n} = x$ is true for all $x, y \in Q$. □

If we know the permutation signature of Q, we would know Q models an identity of the form $((x * \underbrace{y) * y \cdots y) * y}_{n} = x$ if and only if n is a multiple of $lcm(p_1 \ldots p_k)$. Thus, we can avoid doing the actual computation to test the identities if we know the permutation signature of Q.

6.2 Improvement

Although our invariant checking has made a great improvement on `isofilter`, there are still a lot of improvements we can make. Applying `isofilter` to large quandles takes too long to be practical. Separating large quandles into buckets does not make computation much more feasible since the `isofilter` checking between any large non-isomorphic quandles still takes extremely long. Part of the goal for our future research is to filter quandles without using `isofilter` and to improve the performance of invariant checking.

The first possible improvement on identities checking is a dynamic database of identities. Identities checking can then be separated into two parts: a static set of identities that can distinguish most of the isomorphism classes would be stored in database; some other identities of large sizes will be randomly generated during run-time. In addition to the list of effective identities we already have, random large identities would certainly strengthen the invariants checking.

Our random identities generator should be part of a knowledge base of quandle invariants. Given a stream of input quandles, we would first test the identities in the static list on the quandles and separate them into buckets. If there are buckets containing more than one quandle, we will generate certain number of random identities to test on quandles in these buckets. If a random identity is able to reduce the size of some buckets, it should be inserted to our static identity list. As our database of identities grow, our invariant checking will become stronger over time. Furthermore, our knowledge base should contain a list of identified isomorphism classes of quandles. Each isomorphism class should be labeled by the set of identities it models. Such a knowledge base of isomorphism classes and invariants would surely provide a useful tool for knot theorists.

Although it is hard to achieve the lower bound mentioned in Theorem 6.1.1, we should still minimize the number of identities in the static identity list. We should not include any two identities at the same time if they are modeled by the same set of quandles. This idea is basically filtering identities using quandles, and can be implemented with the Prolog database we generate from invariant checking. The following Prolog predicate would allow us to acquire such pairs of identities.

```
isoIdentities(A, B):-identity(A),
                     identity(B),
                     A<B,
                     bagof(X,identitySatisfy(X,A),L),
                     bagof(Y,identitySatisfy(Y,B),L).
```

The program `prover9` within the LADR allows us to check whether two identities are equivalent. For example, we want to see whether identity $(x/y)/x = y$ would imply $((((((x/y)/y) * x) * y)/x)/x)/x = y$ because they are modeled by the same set of quandles up size 30. Then we can write the following into an input file named `formulas.in`

```
formulas(sos).
x*x=x.
(x*y)/y=x.
(x/y)*y=x.
```

```
(x*y)*z = (x*z)*(y*z). %quandle axioms
(x/y)/y=x.   %Identity 1
end_of_list.
formulas(goals).
((((((x/y)/y)*x)*y)/x)/x)/x=y %Identity 2
end_of_list.
```

Then we run the following command line on this file.

```
prover9 -f formulas.in
```

If identity 1 can indeed derive identity 2 along with the quandle axioms, `prover9` would construct a proof. Thus, if we can verify two identities are equivalent using `prover9`, we would remove one of them from the database.

7

Appendix A

In this appendix, we include the list of the buckets that contain more than 1 isomorphism classes. The quandles are numbered by Belk's GAP4 quandle database.

- Size 12:
  - 3, 9
- Size 15
  - 1, 5
- Size 21
  - 1, 8
- Size 24
  - 2, 3
  - 13, 28
  - 14, 15, 31, 33

  - 16, 30, 32
  - 19, 38
  - 21, 27
  - 22, 39

- Size 27
  - 8, 32, 33, 36, 38
  - 27, 28, 31, 34, 35, 37, 39, 42, 43, 48, 49, 50, 51, 52, 53, 57, 58, 59, 60
  - 44, 47, 62
  - 45, 46, 61
  - 54, 55, 56

- Size 28
  - 1, 12
  - 2, 11

- Size 30
  - 1, 9, 21
  - 3, 12

8

Appendix B

The following C code is responsible for computing the permutation signature from a quandle interpretation.

```
/*This method finds the cycle leader*/
int findNextLeader(int *checked, int start, int size){
int index = start +1;
while(index < size && checked[index] == CHECKED) index ++;
if(index >= size) return CHECKED;
else return index;
}

/*This method retrieves the cycle structure from a table*/
void get_permutation(int *table, int size, int icount){
int column[size]; /*this will be a column in a quandle table*/
int checked[size]; /*this will keep track which element can be perm leader*/
int i;
checked[0] = CHECKED; /*The first cycle should be trivial for 0*/
for(i = 1; i< size; i++) checked[i] = NEW;
for(i = 0; i < size; i++) column[i] = table[i * size];

int start = 0;
int index;
printf("permSig(%d , [", icount);
int comma = 0;
while((index = findNextLeader(checked, start, size)) !=
CHECKED){    /*index is the leader of a cycle*/
int next = column[index];
checked[next] = CHECKED;
int count = 1;

while(next != index){
next = column[next];
checked[next] = CHECKED;
count ++;
}
if(comma == 0) printf("%d", count); //Trim the last coma
else printf(", %d", count);
start = index;
comma++;
}
printf("]).\n");
}
```

9

Appendix C

The following Prolog code is the supplement code for Section 4.3. It removes some of the equivalent identities from the list.

```
variable(x).
variable(y).
switch(x,y).
switch(y,x).

%Note: We reverse the variables here
reverse_One(star(E, X), Z, divide(Z, Y)) :- switch(X, Y), variable(E),!.
reverse_One(star(E, X), Z, R):- switch(X,Y), reverse_One(E, divide(Z, Y), R).
reverse_One(divide(E, X), Z, star(Z, Y)) :- switch(X,Y), variable(E),!.
reverse_One(divide(E, X), Z, R) :- switch(X,Y), reverse_One(E, star(Z,Y), R).

%Note: We don’t reverse the variables here
reverse_Two(star(E, X), Z, divide(Z, X)) :- variable(E),!.
reverse_Two(star(E, X), Z, R):- reverse_Two(E, divide(Z, X), R).
reverse_Two(divide(E, X), Z, star(Z, X)) :- variable(E),!.
reverse_Two(divide(E, X), Z, R) :- reverse_Two(E, star(Z,X), R).

%Note: Reverse Variable twice, so get back to X
reverse(star(E,x), R)   :- reverse_One(star(E, x), x , R).
reverse(divide(E,x), R) :- reverse_One(divide(E, x), x, R).
reverse(star(E,y), R)   :- reverse_Two(star(E, y), x, R).
reverse(divide(E,y), R) :- reverse_Two(divide(E,y), x, R).

reverseMember(T1, [T2|_]) :- reverse(T1,T2),!.
reverseMember(Term, [_|T]) :- reverseMember(Term, T).

removeReverse([], []).
removeReverse([H|T], New) :- reverseMember(H, T),
                             removeReverse(T, New).
removeReverse([H|T], [H|New]) :- not(reverseMember(H, T)),
                                 removeReverse(T, New).

termlist(L, N) :- bagof(X, generate_exp(X, N), L1),
                  removeReverse(L1,L).

print_equation(T) :- generate_eq(T, V),
                     print_term_init(T), print(=), print(V).

print_list([], _).
print_list([H|T], N):- print_equation(H), print(' '), print(#),
                       print(label(N)), print(.),
                       print('\n'), M is N+1, print_list(T,M).
```

Bibliography

[1] Eleanor Birrell, *The Knot Quandle* (2009).

[2] Aleksandar Chakarov, *Computing the Typeset of Quandles* (2010).

[3] David Dummit and Richard Foote, *Abstract Algebra*, Vol. 3rd Edition, Wiley, 2003.

[4] John Humphreys, *A Course in Group Theory*, Oxford University Press, 1996.

[5] David Joyce, *A Classifying Invariant of Knots, The Knot Quandle* (1980).

[6] Sam Nelson, *A Polynomial Invariant of Finite Quandles* (2007).

[7] Hannah Quay-De La Vallee, *Introduction to Quay Theory* (2009).